About This Book

This book is designed as a programmer's guide for the IBM SystemView Agent Version 1 program (hereafter referred to as SystemView Agent). SystemView Agent provides access to system components that have been defined according to the Desktop Management Interface (DMI) standard by acting as an SNMP agent. These components can be hardware or software in the system that have been defined in the Management Information Format (MIF). Although the DMI itself is protocol-independent, SystemView Agent can detect any DMI-enabled components in the system and translate the MIF information into SNMP management information bases (MIBs).

This book contains information for versions of the SystemView Agent program that run in the following environments:

- OS/2
- AIX
- Windows NT and Windows 95

Information that is specific to a particular operating system is identified in the text. If no operating system references are included, the information is pertinent to all operating systems that the SystemView Agent program supports.

Who Should Use This Book

You should read this book if you are an agent programmer who will use SystemView Agent. You should use this book if you are doing one or more of the following:

- Developing a management application that uses the DMI
- Enabling a product for DMI technology

Before reading this book, you should have a general understanding of network management, the operating system on which you are working, and the C programming language.

In addition, you should be familiar with the *Desktop Management Interface Specification, Version 1.1* , which is created and distributed by the Desktop Management Task Force.

How to Use This Book

Read Introducing the SystemView Agent and the DMI for an introduction to the SystemView Agent program and a brief overview of the Desktop Management Task Force (DMTF) specification of the DMI. This chapter also outlines the steps involved in developing management applications and developing managed components that use the DMI.

Read Understanding the MIF for information about the structure and conventions of the Management Information Format (MIF) used to describe components to the DMI.

Read Using the DMI Management Interface for information about using the Management Interface (MI) of the DMI. This chapter describes registration with the service layer and status code processing.

Read Using the DMI Component Interface for information about using the Component Interface (CI) of the DMI. This chapter describes installing MIF files, registering component instrumentation with the service layer, and generating indications.

Read DMI Command Blocks for information on the command blocks used to build DMI commands. This chapter describes common command blocks, MI command blocks, and CI command blocks.

Read Enabling a Product for the DMI for information about defining the manageable elements of a product to be managed through the DMI. This chapter discusses how to plan and code the Management Information Format (MIF) file for the new component and how to create program code to manage the component. How to send events to the service layer is also discussed.

Read DMI Procedure Library (DMIAPI) for information about the DMI procedure library supplied with the SystemView Agent. This chapter describes the function calls and data types used to communicate with the DMI.

Read Implementing DMI on OS/2 for information about developing applications and component instrumentation in an OS/2 environment.

Read Implementing DMI on AIX for information about developing applications and component instrumentation in an AIX environment.

Read Implementing DMI on Windows NT/Windows 95 for information about developing applications and component instrumentation in a Windows NT or Windows 95 environment.

This book also contains a glossary, a bibliography, and an index.

Highlighting and Operation Naming Conventions

The following highlighting conventions are used in this book with the noted exceptions:

Bold	Identifies menu choices, pushbuttons, commands, and shell script paths (except in reference information), default values, user selections, daemon paths (on first occurrence), and flags (in parameter lists).
Italics	Identifies parameters whose actual names or values are to be supplied by the user, and terms that are defined in the following text.
`Monospace`	Identifies subjects of examples, messages in text, examples of portions of program code, examples of text you might see displayed, information you should actually type, and examples used as teaching aids.

Terms and Abbreviations

This book uses the following terms:

CI	Component Interface
DMI	Desktop Management Interface
DMTF	Desktop Management Task Force
MI	Management Interface
MIF	Management Information Format
RFC	An abbreviation for Internet Request for Comments documents
SNMP	Simple Network Management Protocol

Where to Find More Information

The SystemView Agent worldwide web page can be accessed through the following uniform resource locator (URL): http://www.software.ibm.com/ sysman/technology/caprod.html. See the web page for current updates and news about the product.

The following publications are included with the SystemView Agent program:

- *SystemView Agent for OS/2 User's Guide* , SVAG-USR2 (OS/2 installations only)
- *SystemView Agent for AIX User's Guide* , SVAG-USRX (AIX installations only)
- *SystemView Agent for Win32 User's Guide* , SVAG-USRW (Windows NT/Windows 95 installations only)
- *SystemView Agent DMI Programmer's Guide* , SVAG-DMIP
- *SystemView Agent DPI Programmer's Guide* , SVAG-DPIP (OS/2 installations only)

The documents included with SystemView Agent are available in softcopy format in the OS/2 and AIX environments. You can use one of the following browsers:

- For OS/2 installations, use the VIEW facility.
- For AIX installations, use the DynaText browser.

SystemView Agent supports the most current DMI specification. At this time, this is DMI Version 1.1. The *Desktop Management Interface Specification, Version 1.1* is available from the Desktop Management Task Force.

Publications relevant to the SystemView Agent SNMP function are:

- RFC1592, which is the SNMP DPI 2.0 RFC.
- RFC1157, which describes SNMP Version 1
- RFC1901 through RFC1910, which describe SNMP Version 2

Other sources of information that can be helpful when using SystemView Agent are listed in Bibliography.

You can request IBM publications from your IBM representative or the IBM branch office serving your region. You can also contact the place where you purchased the SystemView Agent program.

Introducing the SystemView Agent and the DMI

With the Simple Network Management Protocol (SNMP), the SystemView Agent program provides remote management applications with access to DMI-enabled products in a system. Local management applications that can communicate with the DMI can also query and set MIF information for DMI-enabled components.

This book describes the steps necessary to design and implement management applications and manageable products with DMI technology. The information presented here is accompanied with examples of program code to help show how the DMI elements can be implemented. All of the examples used in this book in their complete form are in the the following directories:

- For OS/2 installations, the DMI\EXAMPLES subdirectory within the directory where you installed the SystemView Agent program.
- For AIX installations, the /usr/lpp/sva/samples subdirectory.
- For Windows NT/Windows 95 installations, the DMI\EXAMPLES subdirectory within the directory where you installed the SystemView Agent program.

SystemView Agent and the Desktop Management Interface

The Desktop Management Task Force (DMTF) is a vendor alliance. This alliance was convened to streamline the management of diverse operating systems commonly found in an enterprise. The DMTF includes industry-wide workgroups, which identify the pieces of information that are necessary to manage specific categories of devices.

As part of this effort, the DMTF has published a standard that is called the *Desktop Management Interface Specification* .

The SystemView Agent program provides access to system components that have been defined according to the DMI standard by acting as an SNMP agent. Although the DMI itself is protocol-independent, SystemView Agent can detect any DMI-enabled components in the system and translate the MIF information into SNMP objects that conform to the management information base (MIB) format. This MIF-to-MIB mapping is performed automatically by a DMI subagent provided with SystemView Agent.

What is the Desktop Management Interface?

The Desktop Management Interface (DMI) comprises a set of interfaces and a service layer that mediate between management applications and components residing in a system. The DMI is a free-standing interface that is not tied to any particular operating system or management protocol.

In the context of the DMI, a *component* is a physical or logical element of a system, such as a piece of hardware or software. The SystemView Agent program can access information for components that you have defined for your system or for components that are

predefined by other products. The program code that manages the component directly is referred to as the *component instrumentation* .

Information about a component is defined in a language that is called the Management Information Format (MIF). The MIF file for a component describes all aspects of that component that can be managed by an application. The MIF files for the components in a system are kept in a MIF database, where they can be accessed through the DMI.

Elements of the DMI describes the elements that make up the DMI.

Table 1. Elements of the DMI

DMI ELEMENT	DESCRIPTION
Component Interface (CI)	The CI provides access to component information and enables the component instrumentation to install or remove its associated MIF.
Management Interface (MI)	The MI enables management applications to access component information and register for event notifications.
Service layer	The service layer coordinates requests from management applications to component instrumentation and performs other high-level synchronization between the MI and the CI.

When a new component is introduced to the system, the component installs the component MIF in the MIF database and notifies the service layer that the component is available. The component then notifies any registered management applications of its presence by sending an unsolicited notification, or *event*, to the service layer. The service layer forwards the event to the appropriate management applications in the form of an *indication* .

If a component does have any instrumentation associated with it, the MIF file for the component is composed of static information.

If a component is using instrumentation to provide access to its information, the instrumentation can be one of the following types of programs:

Overlay	Loaded only when the service layer receives a request for information. After the request is satisfied, the overlay program is unloaded.
Direct-interface	Runs continuously as a separate process. When the service layer receives a request, it passes control temporarily to the direct-interface program, which retrieves the information and passes it back to the service layer.

If a management application wants to display or change information for a component, the application registers with the service layer. After registration, the management application can access component information with Get, Set, and List commands. Get and Set commands are used to read and write MIF attributes that pertain to the manageable elements of a component.

The service layer receives requests from the management application through the MI and forwards them to the component instrumentation through the CI. When responses come back to the service layer from the component, the service layer passes them back to the management application through the MI. By using the DMI to transfer information:

- The management application does not need to understand the mechanism by which instrumentation gathers component information
- The instrumentation does not need to understand the component protocol used by the management application.

How Does SystemView Agent Use the DMI?

The SystemView Agent program extends the use of the DMI by providing a MIF conversion utility and a DMI subagent that performs mapping of DMI information into SNMP MIB information. Each MIF file in the system can be translated to an SNMP MIB with the conversion utility and then loaded in a remote SNMP management application. The management application can send requests to the DMI subagent, which in turn converts the SNMP request into a DMI request. The conversion is performed again, in reverse, when the DMI response is

returned. This enables the SNMP management application to participate in active management of DMI-enabled components.

The DMI subagent's ability to perform this MIF-to-MIB mapping enables an SNMP management application to manage any component that implements the DMI.

Understanding the MIF

Before you define the individual attributes that will populate your groups, you must understand the Management Information Format (MIF). The MIF file is an ASCII file that you create with a text editor to define the elements of your component to the DMI. The DMI service layer maintains all installed MIFs in a MIF database.

This section describes the MIF in detail and includes information on the following topics:

- Conventions
- Common statements
- Definitions

MIF Conventions

The MIF has the following conventions to be followed:

- Lexical conventions
- Comments
- Keywords
- Data types
- Constants
- Definition block delimiters
- Language statement

Lexical Conventions

The character sets that can be used with the MIF conform to one of the following standards:

- The International Standards Organization document ISO 8859-1 (Latin Alphabet No. 1)
- The Unicode 1.1 specification

If a MIF is created according to the Unicode specification, the first two octets of the MIF file must be X'0xFE' and X'0xFF'. If the service layer does not detect these values in the MIF file, the service layer treats the file as an ISO 8859-1 MIF.

There are four classes of tokens: keywords, integer constants, literal strings, and separators. Blanks, tabs, newlines, carriage returns and comments (known collectively as *white space*) are ignored except as they serve to separate tokens, such as adjacent keywords and constants.

When interpreting tokens, the MIF is not case sensitive, unless the token is a literal string enclosed within double quote (") characters. Case is retained for the contents of a literal string. Literal strings separated by white space are concatenated and stored as one literal string.

Comments

Comments can be placed anywhere in the MIF file to annotate the file's contents. Two forward slashes (\\) are used to indicate the beginning

of a comment, and the comment continues through the end of the line. Comments are ignored when the MIF file is interpreted.

Keywords

MIF Keywords displays the keywords that can be used in the MIF.

Table 2. MIF Keywords

access	attribute	class
common	component	counter
counter64	date	description
direct-interface	displaystring	dos
end	enum	gauge
group	id	int
integer	integer64	int64
key	language	macos
name	octetstring	os2
path	read-only	read-write
start	storage	string
table	type	unix
unsupported	value	win16
win32	write-only	specific

Data Types

The following data types are supported by the MIF:

integer (or int)	A 32-bit signed integer with no known semantics
integer64 (or int64)	A 64-bit signed integer with no known semantics
gauge	A 32-bit unsigned integer that can increase or decrease. When a gauge reaches its maximum value (2[32]-1), it continues to report the maximum value until the value decreases below the maximum. The speedometer of an automobile is an example of a gauge.
counter	A 32-bit unsigned integer that never decreases. A counter increases to its maximum value (2[32]-1 or 2[64]-1) and reverts to zero when it reaches its maximum value. The odometer of an automobile is an example of a counter.
counter64	A 64-bit unsigned integer that never decreases. A counter increases to its maximum value (2[32]-1 or 2[64]-1) and reverts to zero when it reaches its maximum value.

string(*n*) or displaystring(*n*)	A displayable string of *n* octets. For ISO 8859-1 implementations, 1 octet per character; for Unicode implementations, 2 octets per character. The value *n* represents the maximum number of octets in the string, although the actual number of octets in use might be shorter than this maximum value. The length of the string is stored in the first four octets, which are not included in the value *n*. You do not need to zero-terminate the string as you do in the C and C++ programming languages. The length of the string represents the number of octets in the string, not the number of characters. The maximum value of *n* that the service layer can display is 508.
octetstring(*n*)	A string of *n* octets that might or might not be displayable. The value *n* represents the maximum number of octets in the string, although the actual number of octets in use might be shorter than this maximum value. The length of the string is stored in the first four octets, which are not included in the value *n*. You do not need to zero-terminate the string as you do in the C and C++ programming languages. The length of the string represents the number of octets in the string, not the number of characters.

date	A 28-octet displayable string.
	
	Dates are defined according to the following format:
	
	`yyyymmddHHMMSS.uuuuuu+ooo`
	
	Where:
	
	yyyy — The year
	
	mm — The number of the month
	
	dd — The number of the day of the month
	
	HHMMSS — The hours, minutes, and seconds, respectively
	
	uuuuuu — The number of microseconds
	
	ooo — The offset from universal time, coordinated (UTC) in minutes. If east of UTC, the number is preceded by a plus (+) sign, if west of UTC, the number is preceded by a minus (-) sign,
	
	While the date value occupies only 25 octets, the date is stored as a 28-octet field to account for memory alignment. The last three octets are zeroes (/0).
	
	For example, Wednesday, May 25, 1994, at 1:30:15 PM EDT would be represented as:
	
	`19940525133015.000000-300`
	
	If necessary, values must be padded with leading zeroes to conform to the specified format, as in *05* being used to represent the month of May. If a value is not supplied for a field, each character in the field must be replaced with an asterisk (*).

Constants

Integer values can be specified according to the following format:

Syntax	**Base**
nnn	Decimal
0*nnn*	Octal
0x*nnn* or 0X*nnn*	Hexadecimal

Where *n* represents a digit in the appropriate base.

Literal strings are characters enclosed by double quote characters, such as "This is an example of a string". Literal strings that are separated by white space are concatenated and treated as one string for the purpose of interpreting the MIF file.

The literal escape character is the backslash (\) and is used to enter the following characters:

Sequence	**Character**
\a	Alert (ring terminal bell)
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\"	Double quote
\x*nn*	Hexadecimal bit pattern. If the MIF file is in ISO 8859-1 format, the bit pattern can be one to two digits (\0 to \ff); if the file is in Unicode format, the bit pattern can be one to four digits (\0 to \ffff).
ooo	Octal bit pattern. If the MIF file is in ISO 8859-1 format, the bit pattern can be one to three digits (\0 to \377); if the file is in Unicode format, the bit pattern can be one to six digits (\0 to \177777). bit pattern, octal

---

Block Scope

The definition blocks specified by the MIF are delimited by the start and end keywords. When using **start** and **end**, you must specify another keyword to indicate the type of definition you are creating. Definition Block Keywords in the MIF displays the keywords for the different block definitions in the MIF and indicates where each is used.

Table 3. Definition Block Keywords in the MIF

KEYWORD	USED WITHIN	DESCRIPTION
component	MIF file	Defines a component. All other definition blocks exist within the scope of this block. There can be only one component definition in each MIF file.
path	component	Associates a symbolic string with path names in a specific operating system. Path definitions are usually placed at the top of the MIF file before any group definitions are specified. Path definitions are optional in the MIF file.
group	component	Defines a collection of attributes and is sometimes used as a template row for a table. At least one group definition is required in each MIF file. Refer to ComponentID Group for more information.
attribute	group	Defines a unit of managed data. All attri-

		bute definitions are specified within the scope of a group definition. A group definition must have at least one attribute definition in it.
table	component	Defines one or more instances of a group using a previously defined group. Table definitions are optional in the MIF file.
enum	component or attribute	Defines a list of integer-to-string mappings. Enumeration definitions with an associated name statement can be defined at the component level, and enumeration definitions without a name statement can be defined within the scope of an attribute definition. Although many enumeration definitions can exist at the component level, only one can be defined for an attribute. Enumeration definitions are optional in the MIF file.

Simplified Structure of a MIF File shows a simplified example of the definition structure of a MIF file. Indentation is used to enhance the readability of the file, although it has no effect on the way the MIF file is interpreted by the DMI.

Simplified Structure of a MIF File

```
start component
      start path
      end path
      start enum
      end enum
      start group
            start attribute
                  start enum
                  end enum
            end attribute
      end group
      start table
      end table
end component
```

Language Statement

The language statement is used to describe the human language used in the MIF file. This statement is optional and appears before the **start component** statement.

The syntax for the statement is:

```
language = "language string"
```

language string is a text string that identifies the language, dialect (as territory), and character encoding.

The format of *language string* is:

```
language-code|territory-code|encoding
```

language-code is a two-letter code defined in ISO 639, *territory-code* is a two-letter code defined in ISO 3166, and *encoding* is either `iso8859-1` or `unicode`.

For example, the language string `"fr|CA|iso8859-1"` indicates French Canadian, with ISO 8859-1 (8-bit) encoding.

If any fields are not supplied, they are omitted, but the two vertical bars must appear in the string. The default language string is `"en|US|iso8859-1"`.

The *encoding* field is ignored in the MIF file because the first two bytes of the file determine the encoding. However, the field is used when communicating through the Management Interface (MI).

The language statement can appear only once in the MIF file.

If you translate a MIF file into a local language, translate only literal strings such as names, descriptions, enumeration literals, and any comments in the MIF. Do not translate class strings, language names, or keywords.

Note: The service layer supplied with the SystemView Agent does not support Unicode and generates an error if it encounters a Unicode MIF.

Common Statements

The following statements can be used in most MIF definitions:

- Name statement
- Description statement
- ID statement

There are several statements that are specific to the group and attribute definitions. These statements are described as part of their associated definitions.

Name Statement

The **name** statement is used within a definition to assign a short identifying string to the definition. This statement is required for all definitions in the MIF file, and only one name statement is allowed for each definition. The MIF does not require that name statements be unique, with the exception of those used in enumeration and path definitions. The name statements in these two cases must be unique among those specified in other enumeration and path definitions in the component definition.

The syntax for the name statement is:

```
name = "name string"
```

name string is defined by the component developer. The length of the string must be less than 256 characters.

The content of the name statement is usually used for display to a user through a management application, such as the DMI browser. Users can also change the value of this string by editing the MIF file directly.

Description Statement

The **description** statement is used within a definition to provide information about the element being defined. The description statement can be used within a component, group, or attribute definition, but you cannot specify more than one description statement for a single definition.

The syntax for the description statement is:

```
description = "description string"
```

description string is defined by the component developer.

The content of the description statement is usually used for display to a user through a management application, such as the DMI browser. Users can also change the value of this string by editing the MIF file directly.

ID Statement

The **ID** statement is used within a definition to assign a unique numeric identifier (ID) to the definition. The DMI refers to the ID when naming items at the API level and when mapping to network management protocols. The ID statement is required for the attribute and table definitions and optional for the group definition. The MIF allows only one ID statement for each definition.

Note: The component, path, and enumeration definitions do not use the ID statement. The component ID is assigned to the component by the service layer when the component is installed.

The syntax for the ID statement is:

```
id = n
```

n is defined by the component developer. The value of *n* must be a non-zero 32-bit unsigned integer. The ID must be unique only within the scope of the definition in which it is specified. For example, all the attributes within a particular group must have different ID values, but IDs are not required to be unique across groups.

Because components and management applications use IDs for communication, users are not allowed to change the values of an ID.

Definitions

The MIF uses the following definitions to specify the manageable aspects of a component:

- Component definition
- Path definition
- Enumeration definition
- Group definition
- Attribute definition
- Table definition

Component Definition

The component definition is the highest-level definition specified in the MIF file. All other elements of the MIF file are defined within the scope of the component definition, and there can be only one component definition in a MIF file.

The syntax for the component definition is:

```
start component
      name = "component name"
      [description = "description string"]
      (contents of component definition here)
end component
```

Path Definition

The path definition is used to specify the location of the files used for active management of the component. The MIF file for a component can contain more than one path definition, as long as the name statement for each path definition is unique within the component definition.

The operating system identifiers used in the path definition include **dos**, **macos**, **os2**, **unix**, **win16**, **win32**, **win9x**, and **winnt**. These identifiers are not case sensitive.

If you specify a keyword of **win32**, you are indicating that the component instrumentation referenced in this definition runs on either Windows NT or Windows 95 (or greater). Use of **winnt** or **win9x** indicates that the instrumentation runs *only* on the platform specified by the keyword.

The syntax for the path definition is:

```
start path
      name  = "name identifying instrumentation code"
      win32 = "C:\\directory\\filename.dll"
      os2   = "direct interface"
      dos   = "C:\\directory\\filename.com"
      unix  = "/directory/filename"
end path
```

The keyword direct interface in the example indicates that the component instrumentation code is a continuously running process that is registered with the service layer.

The value of the name statement can be used as part of the attribute definition to invoke the instrumentation code indicated in the path definition. Read Value Statement for more information about how the code is invoked.

Enumeration Definition

The enumeration definition enables you to associate character strings with signed 32-bit integers. The enumerated lists are then used by the component's instrumentation code to pass the integer values through the DMI, enabling the management application to display the corresponding text string.

The syntax of the enumeration definition is:

```
start enum
      name = "enumeration name"
      type = datatype
      vvv = "literal string"
      [xxx = "string literal"]
end enum
```

enumeration name is a unique name within the component.

The integer values specified by *vvv* and *xxx* can appear in the definition in any order. It is not necessary to represent every number between the lowest and highest values in the definition, as long as each value is unique within the same enumeration definition.

You can use multiple enumeration definitions within a single component definition. Enumeration definitions do not have ID or description statements.

Note: If you specify the same string but different integer values for multiple entries in the enumerated list, the Management Information Format always associates the string with the first integer value defined in the enumerated list. If a management application requests the value for a subsequent instance of the string, the value of the first instance of the string is returned.

Group Definition

In a MIF file, groups are used to arrange one or more attributes into logical sets. You can also use groups to represent tables, which are made up of arrays of attributes. You can use multiple group definitions within a single component definition.

The syntax of a group definition is:

```
start group
      name = "group name"
      class = "class string"
      [id = nnn]
      [description = "description string"]
      [key = nnn[,mm]...]
      [pragma = "pragma string"]
      (attribute definitions go here)

end group
```

If you include an ID statement in the group definition, the value of the ID statement must be unique among other groups within the component.

The relationship between the ID statement and the key statement in the group definition affects whether the group is interpreted as a group, a table, or a table template. ID and Key Statements in MIF describes how the two statements can be used.

Table 4. ID and Key Statements in MIF

KEY	ID	MIF INTERPRETATION
No	Yes	The group is interpreted as a simple scalar group. The ID statement identifies the group.
Yes	No	The group represents a template row in a table that is defined later in the MIF file. The value statements on the attribute definitions refer to default values within the row. A table definition can be used to populate the table according to the template. For more information, read Sample Table Definition.
Yes	Yes	The group represents a table that is managed by the component's instrumentation code. Any table definitions you use later in the component can use this definition as a template.

Class Statement

The class statement is used within a group definition to identify the developer of the group and the version information that applies to the group. All groups that share the same value for the class statement must also share the same attribute definitions, including the same type, access, storage, and ID statements. The name, description, and value statements on the attribute definitions can vary. Identical class strings indicate identical group definitions. The conformance provided by the class statement enables management applications to determine the semantics of the group's attributes. Only one class statement is allowed for each group definition.

The syntax of the class statement is:

```
class = "class string"
```

class string generally takes the following form:

```
"defining body|specific name|version"
```

In this convention, *defining body* is the name of the organization, *specific name* identifies the contents of the group, and *version* identifies the version of the group definition. Spaces are significant in this convention, so that `"DMTF|ComponentID|1.0"` is not interpreted as being the same as `"DMTF | ComponentID | 1.0"`.

Although the MIF does not require that any particular convention be used when defining the class statement, any class statements that do not follow this convention generate a warning message during the installation of the MIF file. Because applications and service layers might

use this convention when obtaining information with the List Component command, component developers are encouraged to use this convention.

If you are developing a component that does not use the entire set of attributes defined by a class, specify the unsupported keyword in the definitions for the unused attributes. Read Value Statement for more information about using this keyword.

Key Statement

When the attributes in a group define a row in a table, the group must contain a key statement that identifies the attribute ID used as the index into the table. Attributes that function as keys can be of any data type and identify no more than one instance of a group. Each instance of a group represents a row of a table. Only one key statement is allowed for each group definition.

The syntax of the class statement is:

```
key = n[,m]
```

n is the ID of the attribute that functions as the key for the table. If you use more than one attribute to index a table, specify the values in the key statement as integers separated by commas. When a management application sends a request or when component instrumentation sends a response regarding information in a table, the key values must be sent in the same order as they are listed in the key statement.

Note: If you are using the DMI subagent to translate MIF information into SNMP MIB information, do not specify an attribute with a data type of integer64 (int64) or counter64 as a key. The DMI subagent does not support 64-bit keys.

Pragma Statement

The pragma statement is used to provide additional information related to the group, but the contents of the statement are never queried or acted on by the DMI.

The syntax of the pragma statement is:

```
pragma = "pragmakeyword:value[,pragmakeyword:value]"
```

Note: The only use for this statement defined in DMI 1.1 is to provide the SNMP object identifier (OID) of the group to facilitate mapping of MIF information to SNMP MIB information. For example:

```
pragma = "SNMP:1.2.3.4.5.6"
```

Attribute Definition

An attribute is a piece of data related to a component. You organize your attributes by defining them within one or more group definitions. Each group in the component must have at least one attribute definition, although many attribute definitions can appear within a single group definition.

The syntax of the attribute definition is:

```
start attribute
      name = "attribute name"
      id = nnn
      [description = "description string"]
      type = datatype
      [access = method]
      [storage = storage type]
```

```
      [value = v | * "name" | "enumeration string"]
end attribute
```

The ID statement is required for the attribute definition and must have a value that is unique among all other attribute definitions within the same group.

Type Statement

The type statement describes the storage and semantic characteristics of the attribute and is required in the attribute definition. Only one type statement can be specified in the same attribute definition.

The syntax of the type statement:

```
type = datatype
```

datatype is usually one of the data types defined in Data Types.

You can also specify an enumerated list as the data type for a type statement. In this case, the attribute is stored and treated as a signed 32-bit integer. If you have already defined an enumerated list at the component level, you can indicate it by name in place of *datatype* : for example, `type = "Verify_Type"`

If you are creating a new enumerated list with the type statement, you can define the list inside the type statement:

```
type = start enum
       enumeration definition
       end enum
```

In this situation the enumeration definition does not require a name statement because the enumerated list cannot be referred to outside the scope of this attribute definition. If you specify a name, the value is ignored.

Access Statement

The access statement determines whether the attribute value can be read or written. The access statement is optional and can appear only once in the attribute definition.

The syntax for the access statement is:

```
access = method
```

method can take the following values: read-only, read-write, write-only.

The access statement is not specified, the default access method used for the attribute definition is read-only.

Note: It is highly recommended that keyed values be assigned an access type of read-only.

> If you are using the DMI browser to display a key attribute with an access type other than read-only, do not attempt to change the key attribute's value or set the attribute's value to be the same as that of another key attribute. This can cause unpredictable results.

Storage Statement

The storage statement provides management applications with information about the attribute to assist in optimizing storage requirements.

Only one storage statement is allowed per attribute definition.

The syntax for the storage statement is:

```
storage = type
```

type can be one of the following:

Common	The value of the attribute is typically limited to a small set of possibilities, such as the clock speed of a CPU or the name of a manufacturer.
Specific	The value of the attribute can be one of a large number of different values, such as the serial number of a device or a timestamp value. Values that have a storage type of specific are generally not suited for optimization.

If the storage statement is not specified on the attribute definition, the default storage type for the attribute is specific.

Value Statement

The value statement provides a value for the attribute being defined or a means by which the value can be determined.

The syntax for the value statement is one of the following:

```
value = v
value = "enumeration value"
value = * "name"
value = unsupported
value = unknown
```

The values specified in the value statement include:

v	The value *v* is used for read-only attributes whose values do not change or for read-write values that are managed by the service layer rather than by component instrumentation. The value specified in the statement must conform to the data type of the attribute. For example, literal strings must be specified within double quotes. Write-only attributes cannot specify a value of *v*.
"enumeration value"	The value "enumeration value" can be either a text string or integer value that references a previously specified enumeration definition within the component or within this attribute definition. The type statement for the attribute must be an enumeration definition, as described in Type Statement.
* "name"	This value specifies the symbolic name of the component instrumentation code that is used to read or write the value for this attribute when a request is made to the service layer. The name indicated by this value must have been previously defined in a path definition within the component definition.
unsupported	This value is a reserved keyword and indicates to the service layer that the attribute is not supported for this component.
unknown	This value is a reserved keyword and indicates to the service layer that the attribute is supported but its value is known.

The value statement is required unless you are defining a template for a table, where it is optional. However, if you do provide a value within a template, that value is used as the default value when populating the table. If you do not specify a value for a template, no default value is used.

Populating Tables

In the MIF, a table is composed of an array of group instances. Each instance of a group is a row in the table. Defining a table can be done in one of two ways:

- By specifying a key statement on a group definition. In this case, the values of the attributes within each row of the table are provided by the component instrumentation.
- By specifying the table's values within the MIF file itself.

The table definition actually identifies the data in the table rather than how the data is stored or changed. When you populate a table with a table definition, you identify a previous group definition in the component to be used as a template for the table.

The syntax for the table definition is:

```
start table
      name = "table name"
      id = nnn
      class = "class string"

      { v1[,v2...] }
      [ { vn[, vm...] }
end table
```

The name statement is required for identification. The value of the ID statement must be unique across all other group and table definitions in the component. The class statement identifies the previous group definition that is being used as the template for this table.

For each row in the table, the values are specified between braces and conform to the format described by the value statement (Value Statement). The values are separated by commas and listed from left to right according to the attribute IDs that correspond to the values. The value of the attribute with the lowest ID is listed first. If a value in the list is omitted, the default value for the corresponding attribute is used if a default value is defined in the template. If no default value is defined, specify a value in the table definition. A row with too few commas is treated as a row with the trailing number of requisite commas, and the values specified in the template are used for the remaining attributes in the row.

A row with too few commas generates an error message from the DMI browser.

Sample Table Definition provides an example of how you can use a table definition to specify attribute values.

Sample Table Definition

```
    start table
        name = "sample table"
        id = 2
        class = "group2 class"
        {7,,,,,,,,,}
        {8,8,8,8,6678,"row 8","octet","20011024123515.000000-300",543,8}
        {9,-99,9,9,6767,"row 9","octet","99999999999.999999-999",333,20}
    end table
```

When populating rows within a table, you must provide unique values for the combination of attributes that make up the key. If the service layer is managing a MIF file and the MIF file does not provide unique keys, the service layer rejects the MIF file. As an example, consider a group definition acting as a template, with two keys specified as *first name* and *last name* . If the table is populated by two rows of attributes, it is acceptable for the values of *last name* to be the same, as long as the values for *first name* are different. In other words, the values "John Smith" and "Paul Smith" are acceptable instances of the two keys, but "John Smith" and "John Smith" are not.

A table definition must be defined after the group definition to which it refers. You can specify the same template for multiple table definitions, as long as each table definition has a different ID.

ComponentID Group

The ComponentID group is a standard group that is required for every MIF file. This group provides basic identification of the component and represents the minimum amount of information that a component developer is expected to provide.

The ID for the ComponentID group is 1. If an attribute is not supported by the component or is not applicable to the component, the value for the attribute should be specified by the unsupported keyword.

The value of the class statement for this group is `"DMTF|ComponentID|1.0"`.

The attributes for the group are described in Attributes for ComponentID Group.

Table 5. Attributes for ComponentID Group

NAME	ID	TYPE	ACCESS	STORAGE	DESCRIPTION
"Manufacturer"	1	string (64)	read-only	common	Organization that developed the component
"Product"	2	string (64)	read-only	common	Name of the component or product
"Version"	3	string (64)	read-only	specific	Version for the component
"Serial Number"	4	string (64)	read-only	specific	Serial number of the component
"Installation"	5	date	read-only	specific	Time and date of the last installation of the component on the system
"Verify"	6	integer	read-only	common	Verification level for the component

If you query the value of the Verify attribute, the component instrumentation verifies whether the component is still in the system and operating properly. The instrumentation returns one of the following values:

0	An error occurred; check status code.
1	This component does not exist.
2	The verify is not supported.
3	Reserved
4	This component exists, but the functionality is untested.
5	This component exists, but the functionality is unknown.
6	This component exists and is not functioning correctly.
7	This component exists and is functioning correctly.

Defining Events in the MIF

The definition of events for a component is done in the MIF file with the following groups:

- Event Generation Group
- Event State Group

The event generation group defines the format for events issued by the component. A management application can issue DMI list commands on the attributes in this group and display information about the cause of the event. If appropriate, information about potential solutions to the problem reported by the event can also be displayed.

The event state group defines the format of state-based events, which are issued when the state of the instrumented component changes. For example, a state-based event might be generated when a device enters a problem state or when a problem state is resolved.

Event Generation Group

The event generation group definition serves as a template that a management application can use to associate values received in the

Indication buffer with enumeration display strings. Although values are specified in the group definition to ensure proper processing of the MIF file, the management application accesses the appropriate values directly from the Indication buffer.

The general structure of the event generation group is:

```
start group
   name = "Event Generation"
   class = "EventGeneration|<specific name>|001"
   id = nnn
   key = nnn[,mmm]...
       (attribute definitions here)
end group
```

The string defined in the class statement is unique for each event generation group and is specified in the following format:

```
<defining-body>^^<specific-name-of-associated-group>
```

The contents of the *specific-name-of-associated-group* field indicates the name of the group to which the events defined in the event generation group correspond.

Component developers can define additional attributes to a standard event generation group by using a *proprietary-extension* field that is appended to the class statement. The format for the class statement in this case is:

```
<defining-body>^^<specific-name-of-assoc-grp>^^<proprietary-extension>
```

When specifying a proprietary extension, it is recommended that you use the complete, registered name of your corporate entity. This ensures that the *specific-name* field in the class statement is unique.

When defining multiple event generation groups for a single associated group, place the event generation groups immediately after the associated group in the MIF file and use sequential IDs.

Boolean Definition

The following Boolean definition is used in specifying the value of some attributes in the event generation group:

```
start enumeration
name = "BOOL"
       0 = "False"
       1 = "True"
end enumeration
```

Event Type

The event type attribute indicates the reason that the event was generated. This attribute is required in the event generation group.

The syntax of the event type definition is:

```
name = "event type"
id = 1
description = "The type of event that has occurred."
type = <enumeration>
access = read-only
storage = specific

value = 0
```

The enumeration list is unique to the event generation group being defined and is not specified in this example.

Event Severity

The event severity attribute indicates the category of the event that has been generated. This attribute is required in the event generation group.

The syntax of the event severity definition is:

```
name = "event severity"
id = 2
description = "The severity of this event."
type =  start enumeration
           0x001 = "Monitor"
           0x002 = "Information"
           0x004 = "OK"
           0x008 = "Non-Critical"
           0x00A = "Critical"
           0x00C = "Non-Recoverable"
        end enumeration
access = read-only
storage = specific

value = 0
```

The enumeration list in this definition cannot be changed.

The events with a severity of Monitor and Information are used to provide information about the event but do not indicate the state of the component that generated the event. Severities of OK, Non-Critical, Critical, and Non-Recoverable, however, are state-based.

Monitor	Used for periodic events generated by transaction-oriented components.
Information	Used to indicate change that is not considered a problem. Information events are not periodic or expected.
OK	Used to indicate that the component has entered the normal state, either for the first time (on initialization, for example) or after a problem state (Non-Critical, Critical, Non-Recoverable) has cleared.
Non-Critical	Used to indicate that a problem needs to be corrected, although no time period for correction is associated.
Critical	Used to indicate that a problem needs to be corrected within a specific time period.
Non-Recoverable	Used to indicate that a problem needs to be corrected immediately. Non-Recoverable events indicate serious failure situations.

Event Is State-Based

This attribute indicates whether the component that generates an event is state-based. If the component is state-based, an event is generated whenever the component's state changes or when any problem state is cleared. If the component is not state-based, an event is generated for any condition of interest that develops, but no event is generated when the condition clears.

This attribute is required in the event generation group.

The syntax of the event-is-state-based definition is:

```
name = "event is state-based"
id = 3
description = "The value of this attribute determines "
              "whether the Event being reported is a "
              "state-based Event or not. If the value of "
              "this attribute is TRUE then the Event is state-"
              "based. Otherwise the Event is not state-based."
```

```
type = "BOOL"
access = read-only
storage = common

value = "False"
```

A value of True indicates that the event is state-based.

Event State Key

This attribute identifies a row in the event state group associated with the component definition that contains this event generation group. The value of the current state attribute (Current State) within the specified row indicates the current state of the event. If the generated event is not state-based, the value of the event state key attribute is ignored.

This attribute is required in the event generation group.

The syntax of the event state key definition is:

```
name = "event state key"
id = 4
description = "This attribute holds the key identifying a row of "
              "the Event State group within the Component in "
              "which the event generator group is located. The "
              "Current State attribute within the row contains "
              "the current state of this state-based event. The "
              "current state can be one of the four severities: "
              "OK, Non-Critical, Critical, and Non-Recoverable."
type = integer
access = read-only
storage = specific

value = 0
```

Associated Group

The associated group attribute contains the value of the class statement of the associated group.

This attribute is a keyed attribute and is required in the event generation group.

The syntax of the associated group definition is:

```
name = "associated group"
id = 5
description = "The class string of the group that is associated "
              "with the events defined in this Event Generation "
              "group."
type = string ( <size> )
access = read-only
storage = common

value = "<class string>"
```

If a management application discovers an event generation group, it can determine the associated group by issuing a DmiListFirstComponentCmd with a class filter of `EventGeneration||` and a keylist containing the value of this attribute.

Event System

The event system attribute provides information about the functional system of the product that generated the event. A management application can use the values of this attribute and the event subsystem attribute (Event Subsystem) to construct a simple message describing the event.

This attribute is required in the event generation group.

The syntax of the event system definition is:

```
name = "event system"
id = 6
description = "The major functional aspect of the product "
              "causing the fault."
type = <enumeration>
access = read-only
storage = specific

value = 0
```

The enumeration list is unique to the event generation group being defined and is not specified in this example.

Event Subsystem

The event subsystem attribute provides information about the functional subsystem of the product that generated the event. A management application can use the values of the event system attribute (Event System) and this attribute to construct a simple message describing the event.

This attribute is required in the event generation group.

The syntax of the event subsystem definition is:

```
name = "event subsystem"
id = 7
description = "The minor functional aspect of the product "
              "causing the fault."
type = <enumeration>
access = read-only
storage = specific

value = 0
```

The enumeration list is unique to the event generation group being defined and is not specified in this example.

Event Solution

The event solution attribute describes a possible solution to the problem that caused the event. The contents of this attribute might be recommended actions for the user of a management application. If the severity of the event is Critical, this attribute might also specify a time period for recovery action.

This attribute is optional in the event generation group.

The syntax of the event solution definition is:

```
name = "event solution"
id = 8
description = "A solution to the problem that caused the "
              "event."
type = <enumeration>
access = read-only
storage = specific
```

```
value = 0
```

The enumeration list is unique to the event generation group being defined and is not specified in this example.

Instance Data Present

This attribute indicates to the management application that the second event block within the Indication data structure contains instance-specific data. This information can be used to identify a particular device that caused the event.

This attribute is optional in the event generation group.

The syntax of the instance-data-present definition is:

```
name = "instance data present"
id = 9
description = "Indicates whether the second event block "
              "contains instance-specific data."
type = "BOOL"
access = read-only
storage = specific

value = "False"
```

Vendor Specific Message

This attribute is used to pass displayable string data to a management application. The string can be in any locale, as long as it can be displayed using the ISO 8859-1 character set. This definition does not specify a size limit for the string.

This attribute is optional in the event generation group.

The syntax of the vendor-specific-message definition is:

```
name = "event message"
id = 10
description = "Auxiliary information related to the event."
type = string(<size>)
access = read-only
storage = specific

value = ""
```

This attribute, along with the vendor specific data attribute (Vendor Specific Message), is designed to provide event generators and consumers with an efficient, private interface with which to pass arbitrary information. This can be useful to manufacturers that develop both component instrumentation and management applications.

Vendor Specific Message

This attribute is used to pass arbitrary data to a management application. This definition does not specify a size limit for the octet string.

This attribute is optional in the event generation group.

The syntax of the vendor-specific-data definition is:

```
name = "vendor specific data"
```

```
id = 11
description = "Auxiliary information related to the event."
type = octetstring(<size>)
access = read-only
storage = specific

value = ""
```

This attribute, along with the vendor specific message attribute (Vendor Specific Message), is designed to provide event generators and consumers with an efficient, private interface with which to pass arbitrary information. This can be useful to manufacturers that develop both component instrumentation and management applications.

Event State Group

The event state group is a table that contains the current state of state-based events within the component. The group definition has a key that acts as a unique identifier for each row of the table. A row in the table holds information about a unique event type that is generated from a given event generation group in the component.

An example of the general structure of the event state group is:

```
start group
   name = "Event State"
   class = "DMTF|Event State|001"
   id = nnn
   key = nnn
       (attribute definitions here)
end group
```

You can also maintain the current state information for proprietary state-based events by adding more rows within this group that reference the proprietary event generation group. Refer to Event Generation Group for information on doing this using the event generation group attribute in the event state group definition.

Event Index

The event index attribute specifies the index into the event state table.

The syntax of the event index definition is:

```
name = "event index"
id = 1
description = "A unique index into the Event State table"
type = integer
access = read-only
storage = common
value = 0
```

Event Generation Group

This attribute contains the class string of the event generation group within the component definition that defines the indication format for the related event. The ComponentID of the component with which the event is associated is specified in the header of the Indication data structure.

The syntax of the event-generation-group definition is:

```
name = "event generation group class"
id = 2
description = "The Class String of the event generator group "
              "within the generating Component"
type = string (256)
access = read-only
storage = common
value = 0
```

For each row in the event state group, this attribute identifies the ID of the event generation group that defines the event type represented by the row. A management application can scan for state-based events in a system by using a class filter of `"|EventState|"` to discover instances of the event state group and then scanning the rows of the group for state-based events.

If you are adding rows to the event state group for proprietary state-based events, specify the class string of the proprietary event generation group as the value for this attribute in the additional rows.

Event Type

The event type attribute indicates the reason that the event was generated. The value of this attribute corresponds to an item in the enumeration list specified by the event type attribute in the associated event generation group definition. The specific event generation group is identified by the event generation group attribute (Event Generation Group).

The syntax of the event type definition is:

```
name = "event type"
id = 3
description = "Integer value that identifies one of the "
              "Event types enumerated in the associated Event "
              "Generation group."
type = integer
storage = common
value = 0
```

Current State

The current state attribute indicates the severity of the specific event type represented by this row of the event state group.

The syntax of the current state definition is:

```
name = "current state"
id = 4
description = "The current state of the Event type identified "
              "by Event Type attribute in this row."
type = start enum
          0x0004 = "OK"
          0x0008 = "Non-Critical"
          0x000A = "Critical"
          0x000C = "Non-Recoverable"
       end enum
access = read-only
storage = specific

value = "OK"
```

The enumeration defined in this attribute is a subset of the enumeration defined in the event severity attribute in the event generation group definition.

Associated Group Keys

This attribute is used to identify an instance of the associated group that is likely to have generated the event. Depending on how the associated group is defined, there might be multiple keys, each of a different type. This attribute represents such a list of keys with an encoded string consisting of a list of integers separated by commas, with no spaces.

The syntax of the associated-group-keys definition is:

```
name = "associated group keys"
id = 5
description = "A list of integer keys that identify the instance "
              "of the Associated Group that actually generated "
              "the state-based event. The list of integer keys "
              "are represented in the value of this attribute as "
              "a string containing a comma-separated list of "
              "integers. The management application must parse "
              "this string to obtain the list of integer keys."
type = string(256)
access = read-only
storage = specific

value = ""
```

Using the DMI Management Interface

This chapter describes how a management application communicates with the DMI through the Management Interface (MI). A management application requests information about components in the system by issuing the DMI Get, Set, and List commands. However, before it can manage a component in the system, the application must register with the service layer for that particular system. Once registered, the application can issue commands, as well as receive notifications of indications. It is the responsibility of the management application to have program code in place to process the error codes returned by the service layer.

This chapter describes how a management application communicates with the MI, including the registration process that the application must use and the status codes that the service layer returns.

Invoking a Command through the Management Interface

Because the DMI is a data interface, all commands are specified with data blocks. In order for a management application to send a command block to the service layer, the application builds the command with the DmiMgmtCommand block and any following blocks and then issues the *DmiInvoke()* function call.

The DmiInvoke() function call sends the command to the service layer for processing. The C-language prototype for this call is:

```
unsigned long DmiInvoke(PTR command)
```

command is the complete command block. The return result is a 32-bit status value that indicates whether the command succeeded or failed. The possible status values are described in Status Codes.

While it processes the command, the service layer immediately returns control to the management application. The management application can then continue processing and issue additional commands, but the application cannot re-use the same DmiMgmtCommand buffer that it previously sent to the service layer. Simultaneous commands from a management application must use different DmiMgmtCommand blocks. When the service layer is finished processing the command, it notifies the management application by calling the pConfirmFunc() function that was given when the management application originally registered with the service layer. At that point, the management application can re-use the DmiMgmtCommand block.

The DMI specification does not include a time-out policy. If a management application issues a command and does not appear to get a response, the application should issue a cancel command. Implementing DMI on OS/2 describes how the SystemView Agent incorporates time restrictions on some operations.

For a description of the DmiMgmtCommand block and its contents, read DmiMgmtCommand Command Block.

Canceling a Command

A management application can cancel an outstanding command by using the DmiMgmtCommand block. If the command being cancelled is performing actions on multiple attributes, the command is halted as soon as possible within the targeted command list.

To cancel a command, set the following fields in the DmiMgmtCommand block to the specified values:

Variable	Value
iCommand	0x102 (DmiCancelCmd)
iMgmtHandle	The value originally assigned by the service layer when the application registered. This value identifies the management application.
iCmdHandle	The value that identifies the handle of the outstanding command. The management application must ensure that the value of iCmdHandle is unique among all outstanding commands from that management application.

For a description of the DmiMgmtCommand block and its contents, read DmiMgmtCommand Command Block.

Registering with the Service Layer

Before initiating any management activity through the DMI, a management application must register with the service layer. The registration process enables the service layer to provide services for the management application, such as forwarding indications from components in the system.

The following registration commands are described in the command blocks chapter, according to their command block structure:

- DmiRegisterMgmtReq (DmiRegisterMgmtReq Command Block)
- DmiRegisterCnf (DmiRegisterCnf Command Block)

Because the service layer maintains the entry points for a management application until the application unregisters, the application cannot provide different entry points for different commands. However, if for some reason a management application needs to use multiple entry points to the service layer, the application can do one of the following:

- Register a dispatch routine as its entry point and then manually dispatch reports to the appropriate code.
- Register with the service layer more than once, giving different entry points for each registration. For each registration, the service layer assigns a new management handle, so the appropriate handle can be used when issuing a command. Because each registration adds processing overhead to the service layer, this approach should be used only when the first method is not practical.

In addition to the management application's entry points, the service layer must keep the value of the osLanguage string in the command block. This value identifies the language and character set that the service layer uses when sending indications to the management application.

Unregistering with the Management Interface

To unregister a management application from the service layer, send a DmiMgmtCommand block with the following values:

Variable	Value
iCommand	0x101 (DmiUnregisterMgmtCmd)
iMgmtHandle	The value originally assigned by the service layer when the application registered. This value identifies the management application.

For a description of the DmiMgmtCommand block and its contents, read DmiMgmtCommand Command Block.

Processing Indications

An indication is an unsolicited report that is not expected by the receiving application. Registration provides a mechanism for management applications to register for notification of indications.

The service layer maintains a list of entry points for registered management applications. Each entry point is called when the service layer generates an indication. The called routine must immediately return control to the service layer and perform any operations necessary based on the indication. The routine informs the service layer that it has finished processing the report by calling the function *pResponseFunc()* , as given in the DmiIndicate block.

For a description of the DmiIndicate block and its contents, read DmiIndicate Command Block.

Each registered indication entry point is called for every indication.

Management applications must unregister their entry points before terminating.

Using List Commands

The List commands provide read access to a system's MIF database and enable a management application to query the manageable entities within a system. These commands allow a system to be probed to determine its characteristics without incurring the overhead of executing the code required to retrieve current values. All list commands take the following form:

Form	**Behavior**
List	Accesses lists randomly
List First	Accesses lists from the start
List Next	Accesses lists in sequential order

Specific List commands return information on individual entities. List First and List Next commands return data in a bulk fashion. There are also List commands to retrieve the description fields from a system.

Listing Component Information

List Component commands request a list of the components in a system. This command can be used to return all components in a system or only those with a certain group class. Calls for all components are referred to as *unfiltered*, and calls for a specific group class are *filtered* .

There are three List Component commands:

DmiListComponentCmd
: Requests data for a specific component.

DmiListFirstComponentCmd
: Requests data for the first component in a system. The component with the lowest component ID is considered the first component in the system. This will always be the service layer itself, which always has component ID of 1. The service layer fills in as many component definitions as the size of the confirm buffer allows.

DmiListNextComponentCmd
: Requests data for the component instrumentation whose ID is the next highest value after the one passed in the DmiListComponentCmd block. The service layer fills in as many component definitions as the size of the confirm buffer allows.

For a description of the DmiListComponentReq block used to issue these List commands, read DmiListComponentReq Command Block.

The service layer does not contact any component instrumentation when performing List commands. It relies on the MIF database for all

information. When specifying keys that require components to be run before the values can be determined, the service layer returns the key as a possible match. Any matches from data in the MIF database are listed as hard matches. It is the responsibility of the management application to follow up with each component instrumentation and convert possible matches into hard matches.

Listing Group Information

List Group commands request a list of the groups in a component.

There are three List Group commands:

DmiListGroupCmd
: Requests data for a specific group.

DmiListFirstGroupCmd
: Requests data for the first group in a specified component. The group with the lowest group ID is considered the first group in the component. The service layer fills in as many groups as the size of the confirm buffer allows.

DmiListNextGroupCmd
: Requests data for the group whose ID is the next highest value after the one passed in the DmiListGroupReq block. The service layer fills in as many groups as the size of the confirm buffer allows.

For a description of the DmiListGroupReq block used to issue these List commands, read DmiListGroupReq Command Block.

Listing Attribute Information

List Attribute commands request the attribute definitions within a group for a given component. They do not retrieve the current attribute values.

There are three List Attribute commands:

DmiListAttributeCmd
: Requests data for a specific attribute.

DmiListFirstAttributeCmd
: Requests data for the first attribute in a group. The attribute with the lowest attribute ID is considered the first attribute in the group. The service layer fills in as many attribute definitions as the size of the confirm buffer allows.

DmiListNextAttributeCmd
: Requests data for the attribute whose ID is the next highest value after the one passed in the DmiListAttributeReq block. The service layer fills in as many attribute definitions as the size of the confirm buffer allows.

For a description of the DmiListAttributeReq block used to issue these List commands, read DmiListAttributeReq Command Block.

Listing Description Information

List Description commands request the description field for the component, group, or attribute specified. These calls are separated from the other list calls because the description fields might be of substantial size and might not be used or needed in some cases.

There are three List Description commands:

DmiListComponentDescCmd
: Requests description of components

DmiListGroupDescCmd

	Requests description of groups
DmiListAttributeDescCmd	Requests description of attributes

For a description of the DmiListDescReq block used to issue these List commands, read DmiListDescReq Command Block.

Getting Attribute Information

The difference between List and Get commands is that a List command returns MIF information about an object, and a Get command requests the actual value for the object.

The DmiGetAttributeCmd command is used to request the current values of attributes within groups. For a description of the DmiGetAttributeReq block used to issue this command, read DmiGetAttributeReq Command Block.

Setting Attribute Information

To enable concurrent access by multiple management applications, the DMI allows more than one pass through a list of Set commands. The first pass is a call to reserve the attributes. This reserve command tests a component to verify that a set operation on a given attribute can occur and asks the component instrumentation to validate a set operation without performing the set operation. The component instrumentation must validate the parameters of the call and reserve any resources that would be required if the set command were issued.

A positive response from the reserve command indicates that the component instrumentation has validated the parameters and reserved any resources that might be required to perform the set operation. A negative response from the command indicates that the component instrumentation is unable to perform the set operation.

If all the attributes in the list returned successfully from the reserve operation, the management application typically issues a Set command for each attribute.

If a reserve fails, the management application might want to release each of the attributes it previously reserved. Releasing attributes allows the component instrumentation to deallocate any resources it had acquired. The task of setting attributes back to their previous values is the responsibility of the management application.

There are three Set commands:

DmiSetAttributeCmd	Sets the value for a specified attribute
DmiSetReserveAttributeCmd	Reserves a specified attribute for a subsequent set operation
DmiSetReleaseAttributeCmd	Releases a specified attribute

For a description of the DmiSetAttributeReq block used to issue these Set commands, read DmiSetAttributeReq Command Block.

Reserving Attributes

Reserving an attribute verifies that the set operation is valid. This ensures that a subsequent set operation will be successfully issued. However, it is *not* a locking operation. A management application can never assume exclusive control of a component.

The component instrumentation that receives the Reserve command is responsible for testing the arguments to verify that the set operation can be performed. The testing verifies that the arguments are within range and allocates and reserves any resources that might be needed to ensures the success of a subsequent set operation.

After a successful set operation, the component instrumentation automatically releases any resources that it has reserved. The management application does not need to explicitly issue a Release command after a set.

The component instrumentation requires that all Reserve commands be followed by either Set or Release commands to perform the set operation or to release the reserved resources. The failure of the management application to issue either command is considered a

catastrophic failure and is outside the scope of the component instrumentation's responsibility to try to correct.

If a management application requests that multiple attributes be reserved and one of the reservations fails, the service layer returns the attribute where the command failed. It is the responsibility of the management application to release any successfully reserved attributes.

Releasing Attributes

The Release command is used when the caller wishes to decommit from a set operation after a reserve has been issued.

Getting Attribute Information by Row

Get Row commands request the values of all the attributes in a particular group. Because groups can be a scalar set of attributes or can be arranged as tables-arrays of sets of attributes-it is more convenient to consider them as rows. Groups used as tables include keys to specify the desired instance (or row) of the group.

There are three Get Row commands:

DmiGetRowCmd
: Requests the values of attributes from a specific group or specific row of a table.

DmiGetFirstRowCmd
: Requests the values of the attributes in the first row of a table. This is the same as DmiGetRowCmd for non-tabular groups.

DmiGetNextRowCmd
: Requests the values of the attributes of the row after the row whose key is passed in the command block.

For a description of the DmiGetRowReq block used to issue these Set commands, read DmiGetRowReq Command Block.

> Note for Component Developers Note that the command can be asking for key values or for attribute values. To determine which, the component instrumentation must compare the iGroupKeyCount field of DmiGetRowReq to the iGroupKeyCount field of DmiGetRowCnf. If the key count in the confirm block is less than the key count of the request block, the service layer is requesting that the next key value be returned.
>
> The key value to return for the DmiGetFirstRowCmd command is the value of the next key attribute in the first row of the table. For the DmiGetNextRowCmd command, the key values in the DmiGetRowReq structure should be inspected to determine the current row. Then the value in the succeeding row should be returned.
>
> The DmiGetRowCnf structure is always built by the service layer, and the value of oGroupKeyList specifies the start of DmiGroupKeyData structures. The pCnfBuf member of DmiCiCommand points to the position in the confirm buffer where the key value should be placed. The value of iGroupKeyCount should be incremented, but the iCnfCount member of the DmiMgmtCommand should not.

Status Codes

Status codes are 32-bit unsigned values. The error codes returned by an operating system are not passed back to a management application. Instead, the service layer maps operating system errors into its error range. This insulates management applications from operating system details. Error codes from components are listed in the component's MIF file.

Non-error Condition Codes

Non-Error Condition Codes displays the status codes used to indicate non-error conditions.

Table 6. Non-Error Condition Codes

VALUE	DESCRIPTION
0x00000	Success
0x00001	More data is available

Database Errors

Database Errors displays the status codes used to indicate error conditions related to the MIF database.

Table 7. Database Errors

VALUE	DESCRIPTION
0x00100	Attribute not found
0x00101	Value exceeds maximum size
0x00102	Component instrumentation not found
0x00103	Enumeration error
0x00104	Group not found
0x00105	Illegal keys
0x00106	Illegal to set
0x00107	Cannot resolve attribute function name
0x00108	Illegal to get
0x00109	No description
0x0010a	Row not found
0x0010b	Direct interface not registered
0x0010c	MIF database is corrupt
0x0010d	Attribute is not supported

Service Layer Errors

Service Layer Errors displays the status codes used to indicate error conditions related to the service layer.

Table 8. Service Layer Errors

VALUE	DESCRIPTION
0x00200	Buffer full
0x00201	Ill-formed command
0x00202	Illegal command
0x00203	Illegal handle
0x00204	Out of memory
0x00205	No confirm function
0x00206	No response buffer
0x00207	Command handle is already in use
0x00208	DMI version mismatch
0x00209	Unknown CI registry
0x0020a	Command has been canceled
0x0020b	Insufficient privileges
0x0020c	No access function provided
0x0020d	OS File I/O error
0x0020e	Could not spawn a new task
0x0020f	Ill-formed MIF
0x00210	Invalid file type
0x00211	Service Layer is inactive
0x00212	UNICODE not supported

OS/2 Errors

OS/2 Errors displays the status codes used to indicate error conditions related to the OS/2 operating system.

Table 9. OS/2 Errors

VALUE	DESCRIPTION
0x03000	OS/2 service layer is not initialized
0x03001	Error during IPC creation
0x03002	Error during thread creation
0x03003	Error during queue creation
0x03004	OS/2 service layer terminated

0x03005	Command exception error
0x03006	Error initializing synchronous call
0x03007	Service layer - DLL version mismatch

AIX Errors

AIX Errors displays the status codes used to indicate error conditions related to the AIX operating system.

Table 10. AIX Errors

VALUE	DESCRIPTION
0x04001	An error occurred during socket creation. The specific errno is EAFNOSUPPORT.
0x04002	An error occurred during socket creation. The specific errno is ESOCKTNOSUPPORT.
0x04003	An error occurred during socket creation. The specific errno is EMFILE.
0x04004	An error occurred during socket creation. The specific errno is ENOBUFS.
0x04005	An unknown error occurred during socket creation.
0x04006	An error occurred during socket binding. The specific errno is EBADF.
0x04007	An error occurred during socket binding. The specific errno is ENOTSOCK.
0x04008	An error occurred during socket binding. The specific errno is EADDRNOTAVAIL.
0x04009	An error occurred during socket binding. The specific errno is EADDRINUSE.
0x0400A	An error occurred during socket binding. The specific errno is EINVAL.
0x0400B	An error occurred during socket binding. The specific errno is EACCES.
0x0400C	An error occurred during socket binding. The specific errno is EFAULT.
0x0400D	An unknown error occurred during socket binding.
0x0400E	An error occurred during socket listen call. The specific errno is EBADF.
0x0400F	An error occurred during socket listen call. The specific errno is ECONNREFUSED.

0x04010	An error occurred during socket listen call. The specific errno is ENOTSOCK.
0x04011	An error occurred during socket listen call. The specific errno is EOPNOTSUPP.
0x04012	An unknown error occurred during socket listen call.

Windows NT/Windows 95 Errors

Windows NT/Windows 95 Errors displays the status codes used to indicate error conditions related to the Windows NT and Windows 95 operating systems.

Table 11. Windows NT/Windows 95 Errors

VALUE	DESCRIPTION
0x06000	Windows NT/Windows 95 service layer is not initialized
0x06001	Error during IPC creation
0x06002	Error during thread creation
0x06003	Error during queue creation
0x06004	Windows NT/Windows 95 service layer terminated
0x06005	Command exception error
0x06006	Error initializing synchronous call
0x06007	Service layer - DLL version mismatch

Other Errors

Additional status codes are displayed in Other Errors.

Table 12. Other Errors

VALUE	DESCRIPTION
0x02000	Start of Windows-specific status codes
0x10000	Start of component-specific status codes

Using the DMI Component Interface

This chapter describes how a component communicates with the DMI through the Component Interface (CI) and how component instrumentation registers with the service layer and send indications.

Calls from the service layer through the CI use only attribute, group and key identifiers for naming. By the time the component instrumentation is reached, the service layer has resolved the component ID. Indications sent from the component instrumentation through the CI also need the component ID so that the service layer can identify the component instrumentation.

The mechanics of the communication from the service layer to the component instrumentation are specific to the operating system environment. The entry point function calls that the component instrumentation must respond to are DmiCiInvoke() and DmiCiCancel().

The service layer is responsible for launching the component instrumentation, if necessary, and for passing along the command blocks.

Installing a Component

A component identifies itself to the management system by installing a MIF file into the MIF database. The component installs the MIF file the first time it is introduced to the system. This is the only time that the component needs to perform this installation.

A MIF is installed and removed using the DmiCiInstall and DmiCiUninstall commands. These are the only primitives provided to manipulate the MIF database. There are no MIF database modification mechanisms. If an installed MIF requires modification, it must be removed, modified, and then re-installed.

Upon installation or removal of a MIF file, the service layer issues an indication to all registered management applications.

The installation can optionally include files that provide language mapping and mapping logic between the MIF file and other management information standards.

For a description of the DmiCiInstallData block issued when installing a component, read DmiCiInstallData Command Block.

On successful return, iComponentId is set to a numeric ID assigned by the service layer. This ID uniquely identifies the component described by the installation. This system-specific ID is stored by the service layer in the MIF database. The MIF database is a persistent data store, so the component ID, once assigned, remains the same until the component is removed. The component must remember this component ID, as it is used when issuing indications or when the component is removed from the system.

Upon successful installation, the service layer sends an indication to every management application that is currently registered with it. If the installation created a new component, the value of iIndicationType in the DmiIndicate block is 2. If the installation resulted in any additional MIFs being added to an existing component, the value of iIndicationType is 4. In both cases, the indication data is a DmiListComponentCnf block. The DmiIndicate function call is described in DmiIndicate Command Block.

Removing a Component

Removing a component from the MIF database effectively removes the component from further participation in the management system. In the DMI, removing a component is called *uninstalling* .

Removing a component from the MIF database does not mean the component is not present in the computer system. The component is still in the system, but the management system is not aware of it.

To remove a component from the MIF database, the component instrumentation fills out a DmiCiUninstallData block and sends it to the service layer with the DmiInvoke() function call.

For a description of the DmiCiUninstallData block issued when installing a component, read DmiCiUninstallData Command Block.

Upon successful removal, the service layer sends an indication to every management application that is currently registered with it. The value of iIndicationType in the DmiIndicate block is 3, and the indication data is a DmiListComponentCnf block.

Invoking a Command

The component instrumentation must provide an entry point so the service layer can send command blocks to the component for processing. The service layer uses the DmiCiInvoke() function call to do this. The C-language prototype for this call is:

```
unsigned long  DmiCiInvoke(PTR command)
```

command is the complete command block.

The return result from the component instrumentation is a 32-bit status value indicating success or failure. The possible status values are described in Status Codes

---

Canceling a Command

The service layer can notify the component instrumentation of an intent to cancel an individual command by calling the DmiCiCancel() entry point within the component instrumentation. This can be issued as a result of a management application issuing a DmiCancelCmd command to the service layer. The C-language prototype for this call is:

```
unsigned long  DmiCiCancel(void)
```

No parameters are needed since by definition the component instrumentation can only be processing one command at a time.

The return result from the component instrumentation should be zero for success and non-zero for any error. The service layer will place the result in iStatus when sending the confirm block to the management application.

Registering Instrumentation

Component instrumentation code can register with the service layer. Registration is desirable when the component instrumentation code is already running, such as management code associated with a running device driver. Registering management entry points within the running code is called registering the direct interface. The direct interface is also useful when the time required to locate and load an executable file and set up and terminate a process would be excessive.

Component instrumentation that has registered a direct interface causes the service layer to override its current access mechanism for the registered attributes. Instead of manipulating the data in the MIF database or invoking programs, the service layer calls the entry point provided in the registration call. When the component unregisters, the service layer returns to its usual method of processing requests for the data, as defined in the MIF file. In this way, component instrumentation can temporarily interrupt normal processing to perform some special function. Note that registering attributes through the direct interface overrides attributes that are already being served through the direct interface.

The DmiRegisterCiInd block can be used:

- To register a callable interface for components that have resident instrumentation code
- To get the version of the service layer.

For a description of the DmiRegisterCiInd block issued when registering instrumentation, read DmiRegisterCiInd Command Block.

To unregister component instrumentation, the DmiUnregisterCiInd block is issued to the service layer. For a description of the DmiUnregisterCiInd block issued when unregistering instrumentation, read DmiUnregisterCiInd Command Block.

Sending Events

Component instrumentation can send unsolicited messages to the service layer to notify it of some particular situation. These unsolicited messages are known as events. When they reach a management application they are usually known as indications, with an indication ID of 1. Events are often used to describe a catastrophic occurrence or other activity that a management application should be informed of quickly. The event identification is specific to a given component.

The DmiIndicate() function call is provided so that component instrumentation can send an indication block to the service layer for processing. The C-language prototype for this call is:

```
unsigned long  DmiIndicate(PTR command)
```

command is the complete command block. The return result is a 32-bit status value indicating success or failure. The possible status values are described in Status Codes.

While it processes the indication, the service layer immediately returns control to the component instrumentation. The component instrumentation can then continue processing and issue additional indications, but the instrumentation cannot re-use the same indication buffer that it previously sent to the service layer. Simultaneous indications from a component instrumentation must use different indication blocks. When the service layer is finished processing the indication, it notifies the component instrumentation by calling the pResponseFunc() function that was given in the original indication block. At that point, the component instrumentation can re-use the indication block.

To send an event, the component instrumentation sends a DmiIndicate block to the service layer's DmiIndicate() entry point. The event data is defined in a DmiEventData block that maps onto the oIndicationData field in the DmiIndicate block.

For a description of the DmiIndicate block used to issue an indication, read DmiIndicate Command Block.

If you are using the DMI subagent to translate DMI indications into SNMP traps, note the following when creating indications:

- Keep the length of any strings passed in a DmiIndicate() call to 512 characters or less. Strings longer than this limit are truncated to 512 characters in the corresponding SNMP trap.
- Restrict the size and number of variables associated with each DMI indicate such that the indication will fit within the maximum SNMP frame size supported in your network.

DMI Command Blocks

The commands issued to the DMI by management applications and components are composed of data structures called command blocks. A typical DMI command contains the following:

- A standard command block, such as the DmiMgmtCommand block. This block appears first in the command block and provides a command code that indicates the specific operation to perform and the composition of the remainder of the command block.
- One or more blocks pertaining to a specific command.
- Any data associated with the command-specific blocks.

This chapter describes the command blocks used with the DMI. There are three types of DMI command blocks:

- Common command blocks
- Management Interface (MI) command blocks
- Component Interface (CI) command blocks

Each command block described in this chapter has the following information presented in tabular format:

Offset	Indicates the location of the element in the buffer. The offset is the number of octets from the start of the buffer.
Size	Indicates the size of the element in octets.
Type	Indicates the type of element. Possible types include: INT — Integer OFFSET — Offset to another element in the block PTR — Pointer to a buffer STRUCT — Command block structure TIMESTAMP — Timestamp block
Variable name	Indicates the name of the element

Common Command Blocks

The DMI uses a common set of data blocks to form the basis of the commands that are sent between management applications and component instrumentation. The following common blocks are used as part of request blocks, response blocks, and indication blocks:

- DmiMgmtCommand
- DmiCiCommand
- DmiConfirm
- DmiIndicate
- DmiVersion
- DmiGroupKeyData
- DmiTimeStamp

The code samples in Management Interface (MI) Command Blocks and Component Interface (CI) Command Blocks provide examples of how these common command blocks can be used.

DmiMgmtCommand

All commands issued through the MI start with a DmiMgmtCommand block. Each outstanding command requires a separate DmiMgmtCommand block. Typically, a command consists of the DmiMgmtCommand block, with additional parameters appended to specify such things as the target of a request or the key data for a table, for example.

The format for the command block is:

Table 13. DmiMgmtCommand Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iLevelCheck
4	4	INT	iCommand
8	4	INT	iCmdLen
12	4	INT	iMgmtHandle
16	4	INT	iCmdHandle
20	4	OFFSET	osLanguage
24	4	OFFSET	oSecurity
28	4	INT	iCnfBufLen
32	4	PTR	pCnfBuf
36	4	INT	iRequestCount
40	4	INT	iCnfCount
44	4	INT	iStatus
48	16	STRUCT	DmiCiCommand

Variable Name **Variable Description**

iLevelCheck	A 4-octet integer with a code value of 0x444d3131 and an ISO 8859-1 value of DMI1. This value indicates the revision level of the DMI specification used by the code issuing the command and also functions as a check, so that the receiver can be confident the pointer is pointing to a valid command block.
iCommand	A code specifying which command to execute. The standard command codes are listed in the description of each command.
iCmdLen	Length in octets of the entire command, including the DmiMgmtCommand block, all appended blocks, and any values stored at the end of the buffer for Set Attribute commands.
iMgmtHandle	A value that uniquely identifies the management application. This value is assigned to the management application by the service layer in response to the DmiRegisterMgmtCmd command. This value is ignored when sending the DmiRegisterMgmtCmd command itself.
iCmdHandle	Optional integer set by the management application to a value unique to all other outstanding commands from that management application. The handle can later be used to cancel the command.
osLanguage	Offset to a string indicating the desired human language for the response. The format of the string is specified in Language Statement. If the management application sets this field to zero, the language specified by the first MIF installed for this component is used.
oSecurity	Offset to an optional security credentials block, as defined in DmiSecurity Command Block, If no block is specified, this value is zero. The service layer and component instrumentation can use the information in this block to implement local security policies.
iCnfBufLen	Length of the confirm buffer, in octets of contiguous memory. The management application sets this to inform the service layer of the maximum size of the response. **Note:** The length of the confirm buffer for DmiListComponentCmd commands should be at least 2K to ensure adequate space for the class strings.
pCnfBuf	Pointer to the buffer where the management application wants the confirm response to go. The management application must supply this buffer; it is not provided by the service layer or component instrumentation. The buffer must be a minimum of 512 octets.
iRequestCount	The number of requests in this command. This count does not include the DmiMgmtCommand block itself.
iCnfCount	Number of valid confirm blocks in pCnfBuf. This value is set on the return of the response.
iStatus	Return status of this command. In the case of an error, where iCnfCount contains the number of valid confirms, iStatus refers to the request represented by (iCnfCount + 1). This value is set on the return of the response.
DmiCiCommand	Command block with information for component instrumentation. Read DmiCiCommand Command Block for a description of the DmiCiCommand command block.

The format of the DmiSecurity block is:

Table 14. DmiSecurity Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iSecurityLen
4	4	INT	iSecurityType
8	x	STRUCT	SecurityData

Variable Name	Variable Description
iSecurityLen	Length of the security block, in octets of contiguous memory.
iSecurityType	Type of security block being passed in SecurityData. Standard security blocks are not provided in version 1.0 of the DMI. The service layer can accept non-standard security blocks, if provided. These blocks are defined by the service layer implementation and exist in the range above 0x80000000 (the high bit of the type is set).

SecurityData	A block that contains the details of the security credentials. The size and structure of the security data depend on the value of iSecurityType.

DmiCiCommand

The DmiCiCommand block is included in the DmiMgmtCommand block and provides information for the component instrumentation.

The format for the command block is:

Table 15. DmiCiCommand Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	OFFSET	oCmdListEntry
4	4	INT	iCnfBufLen
8	4	PTR	pCnfBuf
12	4	PTR	pConfirmFunc

Variable Name	Variable Description
oCmdListEntry	Offset to the command to be processed. This is only used for Get and Set Attribute commands. For Get Attribute commands, the value is an offset to the appropriate DmiGetAttributeData block. For Set Attribute commands, the value is an offset to the appropriate DmiSetAttributeData block. For other commands, the value is zero.
iCnfBufLen	Length of the confirm buffer in octets of contiguous space. This is not necessarily the buffer length listed in the DmiMgmtCommand block. The service layer might have supplied a different buffer specifically for this command.
pCnfBuf	Pointer to the confirm buffer supplied by the service layer where the result should be placed. This is not necessarily the buffer pointed to by pCnfBuf in the DmiMgmtCommand block.
pConfirmFunc	Entry point into the service layer where the component instrumentation calls when the command is completed. The C prototype for this call is: `void pConfirmFunc(DmiConfirm *buf)` *buf* points to a DmiConfirm block.

DmiConfirm

The DmiConfirm block is sent as the result of a request message to indicate that processing of the request has completed.

The format for the command block is:

Table 16. DmiConfirm Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iLevelCheck

4	4	PTR	pDmiMgmtCommand
8	4	INT	iStatus

Variable Name	Variable Description
iLevelCheck	A 4-octet integer with a code value of 0x444d3131 and an ISO 8859-1 value of DMI1. This value indicates the revision level of the DMI specification used by the code issuing the command and also functions as a check, so that the receiver can be confident the pointer is pointing to a valid confirmation block.
pDmiMgmtCommand	Pointer to a command block.
iStatus	A status code indicating an error condition or other information.

DmiIndicate

Entities within the DMI can issue unsolicited reports. Component instrumentation can issue events to the service layer. The service layer can issue notifications of component installations and removals to management applications. These unsolicited notifications are called indications. Each sender of an indication fills out a DmiIndicate block. The DmiIndicate block is then passed to the receiver's indication entry point that was obtained during the registration process.

The format for the command block is:

Table 17. DmiIndicate Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iLevelCheck
4	4	INT	iIndicationType
8	4	INT	iCmdLen
12	4	INT	iComponentId
16	28	TIMESTAMP	DmiTimeStamp
44	4	PTR	pResponseFunc
48	4	OFFSET	oIndicationData

Variable Name	Variable Description
iLevelCheck	A 4-octet integer with a code value of 0x444d3131 and an ISO 8859-1 value of DMI1. This value indicates the revision level of the DMI specification used by the code issuing the command and also functions as a check, so that the receiver can be confident the pointer is pointing to a valid indication block.
iIndicationType	A code specifying the indication type. The possible codes are: 1 Event 2 Install 3 Uninstall 4 Language map
iCmdLen	Length of the entire indication block in octets.
iComponentId	The ID of the component instrumentation generating this indication.

	0	The indication was generated by an entity that has not been installed as a component and has no component ID.
	1	The service layer has initiated the indication.
DmiTimeStamp	A DmiTimeStamp block, as described in DmiTimeStamp Command Block. If a component instrumentation initiated the indication and set the time to zero, the service layer adds the time stamp. If the field is non-zero, the service layer does not check the value for accuracy.	
pResponseFunc	A pointer to the entry point that is called when the receiver finishes processing the indication report. Typically this is only for use between component instrumentation and the service layer. The service layer sets this field to zero before sending indications to management applications. Some service layer implementations might require a response to certain indications. In this case the field is not set to zero, and the management application must call the service layer entry point.	
oIndicationData	An offset to a required buffer that provides context for the indication. The meaning of this data is dependent on the indication type. For events, the DmiEventData block is returned. For component installs and removals, a DmiListComponentCnf block is returned.	

DmiVersion

This block describes the version of the service layer implementation. It is returned by the registration calls DmiRegisterMgmtCmd and DmiRegisterCiCmd.

The format for the command block is:

Table 18. DmiVersion Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	OFFSET	osDmiSpecLevel
4	4	OFFSET	osImplDesc

Variable Name	Variable Description
osDmiSpecLevel	Offset to a string that describes the latest level of the DMI specification supported by this service layer.
osImplDesc	Offset to a string that describes the implementation-specific version of the service layer.

DmiGroupKeyData

This block defines a key to be used when indexing into a group that is arranged as a table. The number of DmiGroupKeyData blocks must match the number of integers specified in the key statement of the component definition. For example, if the key statement identifies attributes 4, 7, and 3 as indexes, there must be three DmiGroupKeyData blocks, with the first corresponding to attribute 4, the next corresponding to attribute 7, and the last corresponding to attribute 3.

Out-of-order keys and partial keys are not supported. A key must uniquely identify no more than one instance of a table.

The format for the command block is:

Table 19. DmiGroupKeyData Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iAttributeId
4	4	INT	iAttributeType
8	4	OFFSET	oKeyValue

Variable Name	Variable Description
iAttributeId	The attribute to be used as an index into a table.
iAttributeType	Data type of the attribute. Possible values are: 1 counter 2 counter64 3 gauge 5 integer 6 integer64 7 octet string 8 displaystring or string 11 date
oKeyValue	Offset to the value of the key.

The attribute ID and type are necessary even though the key statement determines the order because the block is used as follows:

- By management applications to specify keys
- By the service layer for returning the group's key structure
- By component instrumentation to return key values

The fields must always be filled, even if they are not needed.

DmiTimeStamp

This block describes the time format used in the DMI. The format of the time block is a 28-octet displayable string with ISO 8859-1 encoding, so each element is one or more printable characters.

DmiTimeStamp Command Block describes the contents of the string:

Table 20. DmiTimeStamp Command Block

OCTETS	CONTENTS	ENCODING
1-4	year	decimal
5-6	month	decimal (1..12)
7-8	day	decimal (1..31)
9-10	hour	decimal (0..23)
11-12	minutes	decimal (0..59)
13-14	seconds	decimal (0..60)
15	dot	"."
16-21	microseconds	decimal (0..999999)
22-25	offset from UTC (in	["+" | "-"] (000..720)

	minutes)	
26-28	unused	(for purposes of alignment)

For example, Wednesday May 25, 1994 at 1:30:15 PM EDT would be represented as:

```
19940525133015.000000-300
```

A seconds value of 60 is used for leap seconds.

The offset from UTC is the number of minutes west (negative number) or east offset from UTC that indicates the time zone of the system.

Values must be zero-padded if necessary, like `05` in the example above. If a value is not supplied for a field, each character in the field must be replaced with asterisk (*) characters.

The service layer is not required to check the contents of this string for validity.

Management Interface (MI) Command Blocks

The following command blocks are used when issuing commands to the Management Interface (MI).

- DmiRegisterMgmtReq
- DmiRegisterCnf
- DmiListComponentReq
- DmiListComponentCnf
- DmiListGroupReq
- DmiListGroupCnf
- DmiListGroupPragmaReq
- DmiListAttributeReq
- DmiListAttributeCnf
- DmiListDescReq
- DmiGetAttributeReq
- DmiGetAttributeCnf
- DmiSetAttributeReq
- DmiGetRowReq
- DmiGetRowCnf

DmiRegisterMgmtReq

Before a management application can initiate any management activity through the DMI, the application must register with the service layer. The command block used for registration is the DmiRegisterMgmtReq block.

The format for the command block is:

Table 21. DmiRegisterMgmtReq Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	64	STRUCT	DmiMgmtCommand
64	4	PTR	pConfirmFunc
68	4	PTR	pIndicationFunc

Variable Name	Variable Description
DmiMgmtCommand	The command block. The value of iCommand is 0x100.
pConfirmFunc	Pointer to a routine in the management application that the service layer calls after completing each command. The C prototype for this call is: `void pConfirmFunc(PTR cmd)` The variable *cmd* points to the command block that has completed. If pConfirmFunc() is set to zero, the management application cannot use DmiInvoke to issue management directives but can still receive indications, if pIndicationFunc() is non-zero.
pIndicationFunc	Pointer to a routine in the management application that the service layer calls to send unsolicited reports, either because of an event or because a component has been removed. If the management application does not want to receive indications, it must set this field to zero. The C prototype for this call is: `void pIndicationFunc(DmiIndicate *buf)` The variable *buf* points to a DmiIndicate block.

Issuing DmiRegisterMgmtReq displays an example of how to issue the DmiRegisterMgmtReq command block to the MI.

Issuing DmiRegisterMgmtReq

```
ULONG IssueReg(void)   // attempt to register with the service layer
{
DMI_RegisterMgmtReq_t    *reg;
ULONG RC;

    /* Register with the Service Layer. */
    reg = (DMI_RegisterMgmtReq_t *)malloc(sizeof(DMI_RegisterMgmtReq_t));
    memset((void *)reg,0,sizeof(DMI_RegisterMgmtReq_t));
    reg->DmiMgmtCommand.iLevelCheck = DMI_LEVEL_CHECK;
    reg->DmiMgmtCommand.iCmdHandle = YOUR_COMMAND_HANDLE;  // set the command counter
    reg->DmiMgmtCommand.iCnfBufLen = 4000UL;               // set the size of the response
    reg->DmiMgmtCommand.pCnfBuf = malloc(4000);   // set up the response buffer
    reg->DmiMgmtCommand.iRequestCount = 1;
    reg->DmiMgmtCommand.iCmdLen = sizeof(DMI_RegisterMgmtReq_t);
    reg->DmiMgmtCommand.iCommand = DmiRegisterMgmtCmd;
    reg->pIndicationFunc = (DMI_FUNC3_OUT) myEventHandler;
    reg->pConfirmFunc = (DMI_FUNC3_OUT) myCallBackFunc;
    if((RC = DmiInvoke((DMI_MgmtCommand_t *)reg)) != SLERR_NO_ERROR) {  // call service layer and register
        free(reg->DmiMgmtCommand.pCnfBuf);
        free(reg);
    }
    return RC;
}
```

DmiRegisterCnf

A confirm message is sent if the pConfirmFunc() call is non-zero. The confirm message contains the DmiRegisterCnf block, as pointed to by pCnfBuf in the DmiMgmtCommand block.

The format for the command block is:

Table 22. DmiRegisterCnf Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	8	STRUCT	DmiVersion

8	4	INT	iDmiHandle

Variable Name	Variable Description
DmiVersion	A block defining the version of the service layer.
iDmiHandle	This value is a unique handle for this management application on this system. The value is placed in the iMgmtHandle for all subsequent commands from this application.

Processing DmiRegisterCnf displays an example of how to handle the DmiRegisterCnf callback command block. For this example, all callbacks that are returned to the management application are processed through a single entry point.

Processing DmiRegisterCnf

```
case DmiRegisterMgmtCmd:
    if(miCommand->iStatus == 0){   // all went well
        Response = (DMI_RegisterCnf_t *)miCommand->pCnfBuf;  // type it first, it's easier
        AppInfo.myHandle = Response->iDmiHandle;
        Work = (DMI_STRING *)((char *)Response + Response->DmiVersion.osDmiSpecLevel);  // point to the
                                                                                        // MIF string
        memset(AppInfo.SpecLevel,0,sizeof(AppInfo.SpecLevel));
        strncpy(AppInfo.SpecLevel,Work->body,Work->length);
        Work = (DMI_STRING *)((char *)Response + Response->DmiVersion.osImplDesc);
        memset(AppInfo.SLDescription,0,sizeof(AppInfo.SLDescription));
        strncpy(AppInfo.SLDescription,Work->body,Work->length);

        // Do whatever your application needs to do here
        // to process the callback.

    }
    else{    // this is an error condition

        // Do whatever your application needs to do here
        // to process the error condition.

    }
    break;
```

DmiListComponentReq

The following List Component commands use the DmiListComponentReq command block:

- DmiListComponentCmd
- DmiListFirstComponentCmd
- DmiListNextComponentCmd

The format for the command block is:

Table 23. DmiListComponentReq Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId
68	4	OFFSET	osClassString
72	4	INT	iGroupKeyCount
76	4	OFFSET	oGroupKeyList

Variable Name	Variable Description
DmiMgmtCommand	The command block. The values of iCommand indicate the command: DmiListComponentCmd 0x200 DmiListFirstComponentCmd 0x201 DmiListNextComponentCmd 0x202
iComponentId	Component ID for the desired component. This field is ignored for DmiListFirstComponentCmd. For DmiListNextComponentCmd, the service layer starts filling the confirm buffer beginning with the component ID that follows the value in this field. To start at the first component, set this field to zero. On return from the command, the service layer updates this field with the ID of the last component in the confirm buffer. The management application can continue issuing DmiListNextComponentCmd commands until the returned iStatus field indicates that there is no more data.
osClassString	Offset to a string that acts as a filter to retrieve data for components that have groups with specific class strings. If a field of the class string is not significant, it can be omitted. For example, if an application is looking for all component ID groups, it uses the string "DMTF|ComponentID|". Because the version string is missing, the service layer considers any version a match, as long as the first two fields match. If no filtering is desired, set this field to zero or to an empty class string ("||").
iGroupKeyCount	The number of elements of type DmiGroupKeyData pointed to by oGroupKeyList. If any keys are given, they are used with the group class for filtering, and the service layer returns a list of the components that have a particular row in a particular table. It is an error to specify keys without specifying a class string.
oGroupKeyList	Offset to a list of blocks of type DmiGroupKeyData.

Issuing DmiListComponentReq displays an example of how to issue the DmiListComponentReq command to the MI.

Issuing DmiListComponentReq

```
ULONG IssueListComp(ULONG ComponentID,SHORT ListType)  // issues the list component command to the SL
{
DMI_ListComponentReq_t *ListComp;
ULONG RC;

    ListComp = (DMI_ListComponentReq_t *)malloc(sizeof(DMI_ListComponentReq_t));
    memset((void *)ListComp,0,sizeof(DMI_ListComponentReq_t));
    ListComp->DmiMgmtCommand.iLevelCheck = DMI_LEVEL_CHECK;
    ListComp->DmiMgmtCommand.iMgmtHandle = YOUR_MGMT_HANDLE;     // set the app handle
    ListComp->DmiMgmtCommand.iCmdHandle = YOUR_COMMAND_HANDLE;   // set the command handle
    ListComp->DmiMgmtCommand.iCnfBufLen = 8000UL;                // set the size of the response buffer
    ListComp->DmiMgmtCommand.pCnfBuf = (void *)malloc(8000UL);   // set up the response buffer
    ListComp->DmiMgmtCommand.iRequestCount = 1;
    ListComp->DmiMgmtCommand.iCmdLen = sizeof(DMI_ListComponentReq_t);
    ListComp->iComponentId = ComponentID;
    switch(ListType){
        case 1:
            ListComp->DmiMgmtCommand.iCommand = DmiListFirstComponentCmd;  // set the command
            break;
        case 0:
            ListComp->DmiMgmtCommand.iCommand = DmiListNextComponentCmd;   // look for next one in list
            break;
        case 10:
            ListComp->DmiMgmtCommand.iCommand = DmiListComponentCmd;
            break;
    }
    if((RC = DmiInvoke((DMI_MgmtCommand_t *)ListComp)) != SLERR_NO_ERROR){   // call the SL and register
        free(ListComp->DmiMgmtCommand.pCnfBuf);   // free up the confirm buffer first
        free(ListComp);                           // then free up the command block
    }
    return RC;
}
```

DmiListComponentCnf

On return from the DmiListComponentReq call, the confirm buffer contains an array of one or more DmiListComponentCnf blocks. The value of iCnfCount in the DmiMgmtCommand block is set to the number of DmiListComponentCnf blocks returned from this call. The value of iStatus indicates whether there was more data than would fit in the confirm buffer. If so, the service layer updates iComponentId in the command buffer, so the management application can re-issue the command and resume getting data from the point where it stopped.

The format of DmiListComponentCnf is:

Table 24. DmiListComponentCnf Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iComponentId
4	4	INT	iMatchType
8	4	OFFSET	osComponentName
12	4	INT	iClassListCount
16	4	OFFSET	oClassNameList
20	4	INT	iFileCount
24	4	STRUCT	DmiFileData[]

Variable Name	Variable Description
iComponentId	Handle assigned by the service layer when the component was installed.
iMatchType	Zero (0) if this is a definite match; one (1) if the service layer could not determine the type. A value of 1 indicates that the key values required access to data that is controlled by component instrumentation (List commands never execute component instrumentation). In this case, the management application must query the component instrumentation directly.
osComponentName	Offset to a string that names the component.
iClassListCount	Count of the DmiClassNameData blocks pointed to by oClassNameList. The list represents the names of each class within this component's MIF file.
oClassNameList	Offset to an array of DmiClassNameData blocks, as defined in DmiClassNameData Block.
iFileCount	Count of the number of files in the MIF database for the component.
DmiFileData[]	List of structures that conform to the format defined in DmiFileData Block.

The format for the DmiClassNameData block is:

Table 25. DmiClassNameData Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iGroupId
4	4	OFFSET	osClassString

Variable Name	Variable Description
iGroupId	The ID of the group that corresponds to osClassString.

osClassString	Offset to a string that is the group's class string.

The format for the DmiFileData block is:

Table 26. DmiFileData Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iFileType
4	4	OFFSET	oFileData

Variable Name	Variable Description
iFileType	The code for the type of data that follows: **Type code** — **Meaning** 0 — Reserved, do not use 1 — Reserved, do not use 2 — MIF file name 3 — MIF file pointer 4 — SNMP mapping file name 5 — SNMP mapping file pointer 0x80000000 — start of implementation-specific types
oFileData	Offset to an area of memory that contains the mapping data. The structure of this data depends on the type in iFileType.

Processing DmiListComponentCnf, DmiListGroupCnf, and DmiListAttributeCnf displays an example of how to handle the DmiListComponentCnf callback command block. For this example, all callbacks that are returned to the management application are processed through a single entry point, and the same set of instructions is also used to process the DmiListGroupCnf and DmiListAttributeCnf callback command blocks.

Processing DmiListComponentCnf, DmiListGroupCnf, and DmiListAttributeCnf

```
case DmiListComponentCmd:
case DmiListGroupCmd:  // used to load up the attribute display dialog
case DmiListAttributeCmd:  // used to load up the attribute display dialog
    if(!miCommand->iStatus){   // we found an attribute...
        AttrBuf = (DMI_ListAttributeCnf_t *)miCommand->pCnfBuf;

        // Do whatever your application needs to do here

    }
    break;
```

Processing DmiListComponentCnf for DmiListFirstComponentCmd and DmiListNextComponentCmd displays an example of how to handle the DmiListComponentCnf callback command block in response to the DmiListFirstComponentCmd and DmiListNextComponentCmd commands. For this example, all callbacks that are returned to the management application are processed through a single entry point.

Processing DmiListComponentCnf for DmiListFirstComponentCmd and DmiListNextComponentCmd

```
case DmiListFirstComponentCmd:
case DmiListNextComponentCmd:
    if(!miCommand->iStatus || (miCommand->iStatus == SLERR_NO_ERROR_MORE_DATA)){
        CompBuf = (DMI_ListComponentCnf_t *)miCommand->pCnfBuf;     // get the base ptr here
        for(Count = 0;Count != miCommand->iCnfCount;Count++,CompBuf++){
            Work = (DMI_STRING *)((char *)miCommand->pCnfBuf + CompBuf->osComponentName); // get component
                                                                                          // name

        // Do whatever your application needs to do here

        }
        if(miCommand->iStatus == SLERR_NO_ERROR_MORE_DATA)  // we need to do a getnext attr here
            IssueListComp(LastAttr,0,miCommand->iCmdHandle);  // issues the list component command
                                                              // to the service layer
    }
    break;
```

DmiListGroupReq

The following List Group commands use the DmiListGroupReq command block:

- DmiListGroupCmd
- DmiListFirstGroupCmd
- DmiListNextGroupCmd

All three List Group commands use the DmiListGroupReq block:

Table 27. DmiListGroupReq Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId
:68	4	INT	iGroupId

Variable Name	Variable Description
DmiMgmtCommand	The command block. The values of iCommand are: DmiListGroupCmd 0x204 DmiListFirstGroupCmd 0x205 DmiListNextGroupCmd 0x206
iComponentId	ID of the desired component.
iGroupId	ID of the desired group within the component identified by iComponentId. In DmiListFirstGroupCmd, this field is ignored. For the DmiListNextGroupCmd command, the service layer starts filling confirm buffer beginning with the group ID that follows the value in this field. To start at the first group, set this field to zero. On return from the command, the service layer updates this field with the ID of the last group put in the confirm buffer. The management application can continue issuing DmiListNextGroupCmd commands until the returned iStatus field indicates that there is no more data.

Issuing DmiListGroupReq displays an example of how to issue the DmiListGroupReq command to the MI.

Issuing DmiListGroupReq

```
ULONG IssueListGroup(ULONG ComponentID,ULONG GroupID,SHORT ListType)
{
DMI_ListGroupReq_t *ListGroup;
ULONG RC;

    ListGroup = (DMI_ListGroupReq_t *)malloc(sizeof(DMI_ListGroupReq_t));
    memset((void *)ListGroup,0,sizeof(DMI_ListGroupReq_t));
    ListGroup->DmiMgmtCommand.iLevelCheck = DMI_LEVEL_CHECK;
    ListGroup->DmiMgmtCommand.iMgmtHandle = YOUR_MGMT_HANDLE;     // set the app handle
    ListGroup->DmiMgmtCommand.iCmdHandle = YOUR_COMMAND_HANDLE;   // set the command Handle
    ListGroup->DmiMgmtCommand.iCnfBufLen = 8000UL;                // set the size of the response buffer
    ListGroup->DmiMgmtCommand.pCnfBuf = (void *)malloc(8000UL);   // set up the response buffer
    ListGroup->DmiMgmtCommand.iRequestCount = 1;
    ListGroup->DmiMgmtCommand.iCmdLen = sizeof(DMI_ListGroupReq_t);
    ListGroup->iComponentId = ComponentID;   // set to the currently selected component
    ListGroup->iGroupId = GroupID;
    switch(ListType){
        case 1:
            ListComp->DmiMgmtCommand.iCommand = DmiListFirstGroupCmd;  // set the command
            break;
```

```
        case 0:
            ListComp->DmiMgmtCommand.iCommand = DmiListNextGroupCmd;   // look for the next one in the list
            break;
        case 10:
            ListComp->DmiMgmtCommand.iCommand = DmiListGroupCmd;       // list just this one
            break;
    }
    if((RC = DmiInvoke((DMI_MgmtCommand_t *)ListComp)) != SLERR_NO_ERROR){  // call the SL and register
        free(ListComp->DmiMgmtCommand.pCnfBuf);                     // free up the confirm buffer
        free(ListComp);                                             // free the command block
    }
    return RC;
}
```

DmiListGroupCnf

On return from the DmiListGroupReq call, the confirm buffer contains an array of one or more DmiListGroupCnf blocks. iCnfCount in the DmiMgmtCommand block is set to the number of DmiListGroupCnf blocks returned from this call. iStatus indicates whether there was too much data than would fit in the confirm buffer. In this case, the service layer updates iGroupId in the command buffer so the management application can re-issue the command to continue getting data from where it left off.

The format of the DmiListGroupCnf block is:

Table 28. DmiListGroupCnf Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iGroupId
4	4	OFFSET	osGroupName
8	4	OFFSET	osClassString
12	4	INT	iGroupKeyCount
16	4	OFFSET	oGroupKeyList

Variable Name	Variable Description
iGroupId	ID of the group.
osGroupName	Offset to a string that names the group.
osClassString	Offset to a string that names the class.
iGroupKeyCount	The number of elements of type DmiGroupKeyData pointed to by oGroupKeyList.
oGroupKeyList	Offset to a list of blocks of type DmiGroupKeyData. Within the DmiGroupKeyData blocks, no key values are returned (oKeyValue is set to zero) because the purpose of the List command is to identify the attributes that comprise the key.

Processing DmiListComponentCnf, DmiListGroupCnf, and DmiListAttributeCnf displays an example of how to handle the DmiListGroupCnf callback command block.

Processing DmiListGroupCnf for DmiListFirstGroupCmd and DmiListNextGroupCmd displays an example of how to handle the DmiListGroupCnf callback command block in response to the DmiListFirstGroupCmd and DmiListNextGroupCmd commands. For this example, all callbacks that are returned to the management application are processed through a single entry point.

Processing DmiListGroupCnf for DmiListFirstGroupCmd and DmiListNextGroupCmd

```
case DmiListFirstGroupCmd:
case DmiListNextGroupCmd:
```

```
    if(!miCommand->iStatus || (miCommand->iStatus == SLERR_NO_ERROR_MORE_DATA)){   // we found a component:
                                                                    //  display it and look for the next one
        GroupBuf = (DMI_ListGroupCnf_t *)miCommand->pCnfBuf;
        for(Count = 0;Count != miCommand->iCnfCount;Count++,GroupBuf++){
            Work = (DMI_STRING *)((char *)miCommand->pCnfBuf + Gr        oupBuf->osGroupName); // get to
                                                                                               // component
                                                                                               // name

            // Do whatever your application needs to do here

            if(AddInfo->KeyCount){   // there is a table here

            // there is a table here, you may want to get that information

            }
        }
        if(miCommand->iStatus == SLERR_NO_ERROR_MORE_DATA)  // we need to do a getnext attr here
            IssueListGroup(AddInfo->Component,LastAttr,0,miCommand->iCmdHandle);    // load up the groups
                                                                                    // for this component
    }
    break;
```

DmiListGroupPragmaReq

This command is used to request the contents of the pragma statement in the requested group's definition, if one exists. On return from this call, the confirm buffer contains the text from the pragma statement associated with the group requested. iCnfCount in the command block is always set to 1.

The format for the command block is:

Table 29. DmiListGroupPragmaReq Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId
68	4	INT	iGroupId
72	4	INT	iOffset

Variable Name	Variable Description
DmiMgmtCommand	The command block. The value of iCommand is 0x20C.
iComponentId	The ID of the desired component.
iGroupId	The ID of the desired group within the component identified by iComponentId.
iOffset	The number of characters into the pragma where the service layer should start fetching the pragma string. Specify zero to start at the beginning of the pragma. On return from the command, the service layer updates this field with the index of the last character put into the confirm buffer and sets iStatus accordingly. If iStatus indicates that there are more characters in the string, the management application can re-issue the call to get more of the pragma text.

Issuing DmiListGroupPragmaReq displays an example of how to issue the DmiListGroupPragmaReq command to the MI.

Issuing DmiListGroupPragmaReq

```
ULONG IssueLoadPragma(ULONG Comp,ULONG Group)
{
DMI_ListGroupPragmaReq_t      *ListPragma;
```

```
ULONG RC;

    ListPragma =
        (DMI_ListGroupPragmaReq_t *)
malloc(sizeof(DMI_ListGroupPragmaReq_t));
    memset((void *)ListPragma,0,sizeof(DMI_ListGroupPragmaReq_t));
    ListDesc->iGroupId = Group;
    ListDesc->iComponentId = Comp;
    ListDesc->iOffset = 0;      // start at the beginning of the
Pragma
    ListDesc->DmiMgmtCommand.iLevelCheck = DMI_LEVEL_CHECK;
    ListDesc->DmiMgmtCommand.iMgmtHandle = YOUR_MGMT_HANDLE;
    ListDesc->DmiMgmtCommand.iCmdHandle = YOUR_COMMAND_HANDLE;
    ListDesc->DmiMgmtCommand.iCnfBufLen = 4000UL;// set size of the
response
    ListDesc->DmiMgmtCommand.pCnfBuf = (void *)malloc(4000UL);  //
set rsp ptr
    ListDesc->DmiMgmtCommand.iRequestCount = 1;
    ListDesc->DmiMgmtCommand.iCmdLen =
sizeof(DMI_ListGroupPragmaReq_t);
    // set the command
    ListDesc->DmiMgmtCommand.iCommand = DmiListGroupPragmaCmd;
    // now we can call the service layer
    if((RC = DmiInvoke((DMI_MgmtCommand_t *)ListPragma)) !=
SLERR_NO_ERROR) {
        free(ListPragma->DmiMgmtCommand.pCnfBuf);
        free(ListPragma);
    }
    return RC;
}
```

DmiListAttributeReq

The following List Attribute commands use the DmiListAttributeReq command block:

- DmiListAttributeCmd
- DmiListFirstAttributeCmd
- DmiListNextAttributeCmd

The format for the command block is:

Table 30. DmiListAttributeReq Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId
68	4	INT	iGroupId
72	4	INT	iAttributeId

Variable Name	**Variable Description**	
DmiMgmtCommand	The command block. The values of iCommand are:	
	DmiListAttributeCmd	0x208
	DmiListFirstAttributeCmd	0x209
	DmiListNextAttributeCmd	0x20a
iComponentId	The ID of the desired component.	
iGroupId	The ID of the desired group within the component identified by iComponentId.	

iAttributeId	The ID of the desired attribute within the group. In DmiListFirstAttributeCmd, this field is ignored. For the DmiListNextAttributeCmd command, the service layer starts filling the confirm buffer beginning with the attribute ID that follows the value in this field. To start at the first attribute, set this field to zero. On return from the command, the service layer updates this field with the ID of the last attribute put in the confirm buffer. The management application can continue issuing DmiListNextAttributeCmd commands until the returned iStatus field indicates that there is no more data.

Issuing DmiListAttributeReq displays an example of how to issue the DmiListAttributeReq command to the MI.

Issuing DmiListAttributeReq

```
ULONG IssueListAttributes(ULONG ComponentID,ULONG GroupID, ULONG AttribID,SHORT CommandType)
{
DMI_ListAttributeReq_t *ListAttr;
ULONG RC;

    ListAttr = (DMI_ListAttributeReq_t *)malloc(sizeof(DMI_ListAttributeReq_t));
    memset((void *)ListAttr,0,sizeof(DMI_ListAttributeReq_t));
    ListAttr->DmiMgmtCommand.iLevelCheck = DMI_LEVEL_CHECK;
    ListAttr->DmiMgmtCommand.iMgmtHandle = YOUR_MGMT_HANDLE;          // set the app handle
    ListAttr->DmiMgmtCommand.iCmdHandle =  YOUR_COMMAND_HANDLE;       // set the command counter
    ListAttr->DmiMgmtCommand.iCnfBufLen = 8000UL;                     // set the size of the response
    ListAttr->DmiMgmtCommand.pCnfBuf = (void *)malloc(8000UL);        // set up the response buffer
    ListAttr->DmiMgmtCommand.iRequestCount = 1;
    ListAttr->DmiMgmtCommand.iCmdLen = sizeof(DMI_ListAttributeReq_t);
    ListAttr->iComponentId = ComponentID;   // set to the currently selected component
    ListAttr->iGroupId     = GroupID;       // set to the currenly selected group
    ListAttr->iAttributeId = AttribID;
    switch(CommandType){    // this is the type of command to issue
        case 1:
            ListAttr->DmiMgmtCommand.iCommand = DmiListFirstAttributeCmd;  // set the command
            break;
        case 0:
            ListAttr->DmiMgmtCommand.iCommand = DmiListNextAttributeCmd;   // look for next one in list
            break;
        case 10:
            ListAttr->DmiMgmtCommand.iCommand = DmiListAttributeCmd;    // get this specific one
            break;
    }
    if((RC = DmiInvoke((DMI_MgmtCommand_t *)ListAttr)) != SLERR_NO_ERROR){   // call the SL and register
        free(ListAttr->DmiMgmtCommand.pCnfBuf);
        free(ListAttr);
    }
    return RC;
}
```

DmiListAttributeCnf

On return from the DmiListAttributeReq call, the confirm buffer contains an array of one or more DmiListAttributeCnf blocks. iCnfCount in the DmiMgmtCommand block is set to the number of DmiListAttributeCnf blocks returned from this call. iStatus indicates whether there was too much data than would fit in the confirm buffer. In this case, the service layer updates iAttributeId in the command buffer so the management application can re-issue the command to continue getting data from where it left off.

The format of the DmiListAttributeCnf block is:

Table 31. DmiListAttributeCnf Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iAttributeId
4	4	OFFSET	osAttributeName

8	4	INT	iAttributeAccess
12	4	INT	iAttributeType
16	4	INT	iAttributeMaxSize
24	4	INT	iEnumListCount
28	4	OFFSET	oEnumList[]

Variable Name	Variable Description
iAttributeId	The ID of the attribute.
osAttributeName	Offset to a string that names the attribute.
iAttributeAccess	Access information for this attribute. The three least significant bits indicate the read/write status and can take the values: 0 Unknown (usually indicates an error in the MIF database) 1 Read-only 2 Read-write 3 Write-only 4 Unsupported The database storage hint is encoded in the most significant bit and can take the following values are: 0 Specific (read-write information + 0) 1 Common (read-write information + 0x80000000)
iAttributeType	Data type. Possible values are: 0 Unknown (usually indicates a MIF database error) 1 Counter 2 Counter64 3 Gauge 5 Integer 6 Integer64 7 Octet string 8 Displaystring, string 11 Date
iAttributeMaxSize	Maximum size of the attribute. If the size is unknown, this variable returns a value of -1. For attributes, the size is in the units specified by iAttributeType.
iEnumListCount	The number of entries in the enumerations list.
oEnumList[]	Offset to a list of DmiEnumData blocks, as defined in DmiEnumData Command Block.

The format of the DmiEnumData block is:

Table 32. DmiEnumData Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iEnumValue
4	4	OFFSET	osEnumName

Variable Name	Variable Description
iEnumValue	The integer value of the enumeration.
osEnumName	Offset to a literal string corresponding to the enumeration value.

Processing DmiListComponentCnf, DmiListGroupCnf, and DmiListAttributeCnf displays an example of how to handle the

DmiListAttributeCnf callback command block.

Processing DmiListAttributeCnf for DmiListFirstAttributeCmd and DmiListNextAttributeCmd displays an example of how to handle the DmiListAttributeCnf callback command block in response to the DmiListFirstAttributeCmd and DmiListNextAttributeCmd commands. For this example, all callbacks that are returned to the management application are processed through a single entry point.

Processing DmiListAttributeCnf for DmiListFirstAttributeCmd and DmiListNextAttributeCmd

```
case DmiListFirstAttributeCmd:
case DmiListNextAttributeCmd:
    if(!miCommand->iStatus || (miCommand->iStatus == SLERR_NO_ERR    OR_MORE_DATA)){  // found attribute
        AttrBuf = (DMI_ListAttributeCnf_t *)miCommand->pCnfBuf;
        for(Count = 0;Count != miCommand->iCnfCount;Count++,AttrBuf++){
            Work = (DMI_STRING *)((char *)miCommand->pCnfBuf + AttrBuf->osAttributeName);   // get to the
                                                                                            // component name

        // Do whatever your application needs to do here

        }
        if(miCommand->iStatus == SLERR_NO_ERROR_MORE_DATA)  // we need to do a getnext attr here
            IssueListAttributes(((DMI_ListAttributeReq_t *)miCommand)->iComponentId,
                ((DMI_ListAttributeReq_t *)miCommand)->iGroupId,LastAttr,0,(ULONG)miCommand->iCmdHandle);
                                                                                            // load up the
                                                                                            // attributes
    }
    break;
```

DmiListDescReq

The following List Description commands use the DmiListDescReq command block:

- DmiListComponentDescCmd
- DmiListGroupDescCmd
- DmiListAttributeDescCmd

The format for the command block is:

Table 33. DmiListDescReq Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId
68	4	INT	iGroupId
72	4	INT	iAttributeId
76	4	INT	iOffset

Variable Name	Variable Description
DmiMgmtCommand	The command block. The values of iCommand are: DmiListComponentDescCmd 0x203 DmiListGroupDescCmd 0x207 DmiListAttributeDescCmd 0x20b
iComponentId	The ID of the desired component.
iGroupId	The ID of the desired group within the component identified by iComponentId. This field is

	ignored when listing component descriptions.
iAttributeId	The ID of the desired attribute within the group. This field is only required for getting attribute descriptions. Otherwise, it is ignored.
iOffset	Number of characters into the description where the service layer should start getting the description string. Use zero to start at the beginning of the description. On return, the service layer updates this field with the index of the last character written into the confirm buffer and sets iStatus accordingly. If iStatus indicates that there are more characters to be received, the application can re-issue the call to get more of the description text.

On return from this call, the confirm buffer contains the text from the description associated with the entity requested. In the command block, iCnfCount is always set to 1.

Issuing DmiListDescReq displays an example of how to issue the DmiListDescReq command to the MI.

Issuing DmiListDescReq

```
ULONG IssueLoadDescription(ULONG Comp,ULONG Group,ULONG Attr,ULONG Command)
{
DMI_ListDescReq_t      *ListDesc;
ULONG RC;

    ListDesc = (DMI_ListDescReq_t *)malloc(sizeof(DMI_ListDescReq_t));
    memset((void *)ListDesc,0,sizeof(DMI_ListDescReq_t));
    ListDesc->iAttributeId = Attr;
    ListDesc->iGroupId = Group;
    ListDesc->iComponentId = Comp;
    ListDesc->iOffset = 0;      // start at the beginning of the description
    ListDesc->DmiMgmtCommand.iLevelCheck = DMI_LEVEL_CHECK;
    ListDesc->DmiMgmtCommand.iMgmtHandle = YOUR_MGMT_HANDLE;          // set the app handle
    ListDesc->DmiMgmtCommand.iCmdHandle = YOUR_COMMAND_HANDLE;        // set the command counter
    ListDesc->DmiMgmtCommand.iCnfBufLen = 4000UL;                     // set the size of the response
    ListDesc->DmiMgmtCommand.pCnfBuf = (void *)malloc(4000UL);        // set up the response buffer
    ListDesc->DmiMgmtCommand.iRequestCount = 1;
    ListDesc->DmiMgmtCommand.iCmdLen = sizeof(DMI_ListDescReq_t);
    ListDesc->DmiMgmtCommand.iCommand = Command;  // set the command:
                                                  //      DmiListComponentDescCmd
                                                  //      DmiListGroupDescCmd
                                                  //      DmiListAttributeDescCmd
    if((RC = DmiInvoke((DMI_MgmtCommand_t *)ListDesc)) != SLERR_NO_ERROR) {   // ask for the description
        free(ListDesc->DmiMgmtCommand.pCnfBuf);
        free(ListDesc);
    }
    return RC;
}
```

Processing the Callback from DmiListDescReq displays an example of how to process the callback from the the DmiListDescReq command.

Processing the Callback from DmiListDescReq

```
case DmiListComponentDescCmd:   // the list component command
case DmiListAttributeDescCmd:   // this is the description command for the attribute
case DmiListGroupDescCmd:       // list the group description
    if(!miCommand->iStatus){   // we've gotten a good return code back
        Work = (DMI_STRING *)miCommand->pCnfBuf;
        Working = malloc(Work->length + 1);
        memcpy(Working,Work->body,Work->length);

        // Do whatever your application needs to do here

        free(Working);
    }
    break;
```

DmiGetAttributeReq

The DmiGetAttributeCmd command requests the current values of attributes within groups by using the DmiGetAttributeReq block.

The format for the command block is:

Table 34. DmiGetAttributeReq Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId
68	16	STRUCT	DmiGetAttributeData[]

Variable Name	Variable Description
DmiMgmtCommand	The command block. The value of iCommand is 0x300.
iComponentId	The ID of the desired component.
DmiGetAttributeData[]	One or more DmiGetAttributeData blocks. The calling application must indicate the number of appended blocks in the iRequestCount element of the DmiMgmtCommand block. The DmiGetAttributeData block is defined in DmiGetAttributeData Block.

The format for the DmiGetAttributeData block is as follows:

Table 35. DmiGetAttributeData Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iGroupId
4	4	INT	iGroupKeyCount
8	4	OFFSET	oGroupKeyList
12	4	INT	iAttributeId

Variable Name	Variable Description
iGroupId	The ID of the desired group.
iGroupKeyCount	The number of DmiGroupKeyData blocks pointed to by oGroupKeyList.
oGroupKeyList	Offset to a list of DmiGroupKeyData blocks. iGroupKeyCount is the number of blocks in the list.
iAttributeId	The ID of the desired attribute.

Issuing DmiGetAttributeReq displays an example of how to issue the DmiGetAttributeReq command to the MI.

Issuing DmiGetAttributeReq

```
ULONG IssueLoadAttribute(ULONG CompID, ULONG GroupID, ULONG AttrID)  // queries the service layer for
                                                                     // the attribute value
{
DMI_GetAttributeReq_t  *ListComp;
DMI_GroupKeyData_t *NewKey;
ULONG RC;

    ListComp = (DMI_GetAttributeReq_t *)malloc(4000L);    // allocate a big block, in case we have keys
    memset((void *)ListComp,0,sizeof(DMI_GetAttributeReq_t));
    ListComp->DmiMgmtCommand.iLevelCheck = DMI_LEVEL_CHECK;
```

```
    ListComp->DmiMgmtCommand.iCommand = DmiGetAttributeCmd;  // set the command
    ListComp->DmiMgmtCommand.iCmdLen = 4000L;
    ListComp->DmiMgmtCommand.iMgmtHandle = YOUR_MGMT_HANDLE;            // set the app handle
    ListComp->DmiMgmtCommand.iCmdHandle = YOUR_COMMAND_HANDLE;          // set the command counter
    ListComp->DmiMgmtCommand.iCnfBufLen = 4000UL;               // set the size of the response
    ListComp->DmiMgmtCommand.pCnfBuf = (void *)malloc(4000UL);   // set up the response buffer
    memset(ListComp->DmiMgmtCommand.pCnfBuf,0,4000UL);     // clean out this thing
    ListComp->DmiMgmtCommand.iRequestCount = 1;
    ListComp->iComponentId = CompID;   // set to the currently selected component
    ListComp->DmiGetAttributeListÝ0¨.iGroupId = GroupID;
    ListComp->DmiGetAttributeListÝ0¨.iAttributeId = AttrID;
    if(KEY_COUNT){  // there is a key list
        ListComp->DmiGetAttributeListÝ0¨.iGroupKeyCount = KEY_COUNT;
        NewKey = (DMI_GroupKeyData_t *)((BYTE *)ListComp + sizeof(DMI_GetAttributeReq_t));  // this is the
                                                                                            // start of the keylist
        ListComp->DmiGetAttributeListÝ0¨.oGroupKeyList =
                  (DMI_OFFSET)((DMI_OFFSET)NewKey - (DMI_OFFSET)ListComp);

        // Encode the key list here

    }
    if((RC = DmiInvoke((DMI_MgmtCommand_t *)ListComp)) != SLERR_NO_ERROR){   // call the SL and register
        free(ListComp->DmiMgmtCommand.pCnfBuf);   // free up the response buffer
        free(ListComp);                           // free the command block
    }
    return RC;
}
```

DmiGetAttributeCnf

On return from the DmiGetAttributeReq call, the confirm buffer contains an array of one or more DmiGetAttributeCnf blocks. iCnfCount in the command block lists the number of DmiGetAttributeCnf blocks.

The format for the command block is:

Table 36. DmiGetAttributeCnf Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iGroupId
4	4	INT	iAttributeId
8	4	INT	iAttributeType
12	4	OFFSET	oAttributeValue

Variable Name	Variable Description
iGroupId	The ID of the group to which the attribute belongs.
iAttributeId	The ID of the attribute.
iAttributeType	Data type. Possible values are: 0 unknown (usually indicates a MIF database error) 1 counter 2 counter64 3 gauge 5 integer 6 integer64 7 octet string 8 displaystring, string

	11	date

oAttributeValue	Offset to the value returned from this get operation. If no value is provided (in the case of write-only attributes, for example), the offset is zero.

Processing DmiGetAttributeCnf displays an example of how to handle the DmiGetAttributeCnf callback command block. For this example, all callbacks that are returned to the management application are processed through a single entry point.

Processing DmiGetAttributeCnf

```
case DmiGetAttributeCmd:     // get the attribute value from the SL...
    if(!miCommand->iStatus){   // good status
        AttrVal = (DMI_GetAttributeCnf_t *)miCommand->pCnfBuf;
        Work = (DMI_STRING *)((char *)AttrVal + AttrVal->oAttributeValue);   // get to the component name

        // Do whatever your application needs to do here

    }
    break;
```

DmiSetAttributeReq

The following Set commands use the DmiSetAttributeReq command block:

- DmiSetAttributeCmd
- DmiSetReserveAttributeCmd
- DmiSetReleaseAttributeCmd

The format for the command block is:

Table 37. DmiSetAttributeReq Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId
68	20	STRUCT	DmiSetAttributeData[]

Variable Name	Variable Description
DmiMgmtCommand	The command block. The values of iCommand are: DmiSetAttributeCmd 0x301 DmiSetReserveAttributeCmd 0x302 DmiSetReleaseAttributeCmd 0x303
iComponentId	The ID of the component to be operated on.
DmiSetAttributeData[]	One or more DmiSetAttributeData blocks. The calling application must indicate the number of appended blocks in the iRequestCount element of the DmiMgmtCommand block. The DmiSetAttributeData block is defined in DmiSetAttributeData Block.

The format for the DmiSetAttributeData block is as follows:

Table 38. DmiSetAttributeData Block

OFFSET	SIZE	TYPE	VARIABLE NAME

0	4	INT	iGroupId
4	4	INT	iGroupKeyCount
8	4	OFFSET	oGroupKeyList
12	4	INT	iAttributeId
16	4	OFFSET	oAttributeValue

Variable Name	Variable Description
iGroupId	The ID of the desired group.
iGroupKeyCount	The number of DmiGroupKeyData blocks pointed to by oGroupKeyList.
oGroupKeyList	Offset to a list of DmiGroupKeyData blocks. iGroupKeyCount is the number of blocks in the list.
iAttributeId	The ID of the desired attribute.
oAttributeValue	Offset to the value to be set.

There is no data returned from this call in the confirm buffer. The value of iStatus indicates either success or failure. In the case of failure, the value of (iCnfCount + 1) corresponds to the number of the command block that caused the error.

Issuing DmiSetAttributeReq displays an example of how to issue the DmiSetAttributeReq command to the MI.

Issuing DmiSetAttributeReq

```
ULONG IssueSetAttribute(ULONG CompID,ULONG GroupID, ULONG AttrID,char *Value,USHORT Len)  // issues a set
                                                                                          // to the SL
{
DMI_SetAttributeReq_t  *ListComp;
DMI_GroupKeyData_t *NewKey;
ULONG RC;

    ListComp = (DMI_SetAttributeReq_t *)malloc(4000L);    // allocate a big block, in case we have keys
    memset((void *)ListComp,0,sizeof(DMI_SetAttributeReq_t));
    ListComp->DmiMgmtCommand.iLevelCheck = DMI_LEVEL_CHECK;
    ListComp->DmiMgmtCommand.iMgmtHandle = YOUR_MGMT_HANDLE;          // set the app handle
    ListComp->DmiMgmtCommand.iCmdHandle = YOUR_COMMAND_HANDLE;        // set the command counter
    ListComp->DmiMgmtCommand.iCnfBufLen = 4000UL;                     // set the size of the response
    ListComp->DmiMgmtCommand.pCnfBuf = (void *)malloc(4000UL);        // set up the response buffer
    ListComp->DmiMgmtCommand.iRequestCount = 1;
    ListComp->DmiMgmtCommand.iCmdLen = 4000L;                            e number...
    ListComp->iComponentId = CompID;   // set to the currently selected component
    ListComp->DmiMgmtCommand.iCommand = DmiSetAttributeCmd;  // set the command
    ListComp->DmiSetAttributeListÝ0¨.iGroupId = GroupID;
    ListComp->DmiSetAttributeListÝ0¨.iAttributeId = AttrID;
    ListComp->DmiSetAttributeListÝ0¨.oAttributeValue = sizeof(DMI_SetAttributeReq_t);
    memcpy((char *)((BYTE *)ListComp + sizeof(DMI_SetAttributeReq_t)),Value,Len);
    if(KEY_COUNT){  // there is a key list
        ListComp->DmiSetAttributeListÝ0¨.iGroupKeyCount =  KEY_COUNT;
        NewKey = (DMI_GroupKeyData_t *)((BYTE *)ListComp + sizeof(DMI_SetAttributeReq_t) + Len); // this is
                                                                                                 // the start of the keylist
        ListComp->DmiSetAttributeListÝ0¨.oGroupKeyList = (DMI_OFFSET)((BYTE *)NewKey - (BYTE *)ListComp);

        // Encode the key list here...

    }
    if((RC = DmiInvoke((DMI_MgmtCommand_t *)ListComp)) != SLERR_NO_ERROR) {  // call SL and register
        free(ListComp->DmiMgmtCommand.pCnfBuf);
        free(ListComp);
    }
    return RC;
}
```

DmiGetRowReq

The following Get commands use the DmiSetAttributeReq command block:

- DmiGetRowCmd
- DmiGetFirstRowCmd
- DmiGetNextRowCmd

The format for the command block is:

Table 39. DmiGetRowReq Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId
68	4	INT	iGroupId
72	4	INT	iGroupKeyCount
76	4	OFFSET	oGroupKeyList
80	4	INT	iAttributeId

Variable Name	Variable Description
DmiMgmtCommand	The command block. The values of iCommand are: DmiGetRowCmd 0x304 DmiGetFirstRowCmd 0x305 DmiGetNextRowCmd 0x306
iComponentId	The ID of the desired component.
iGroupId	The ID of the desired group.
iGroupKeyCount	The number of DmiGroupKeyData blocks pointed to by oGroupKeyList.
oGroupKeyList	Offset to a list of DmiGroupKeyData blocks. iGroupKeyCount is the number of blocks in the list.
iAttributeId	The attribute ID just before the attribute ID where the Get Row command should start. Use zero to start at the first attribute in the group. This is generally used when the confirm buffer is too small and the service layer has more information than will fit in the buffer. Upon return, the service layer updates this field with the last attribute returned and sets iStatus to 1 to indicate that there is more information than the confirm buffer could hold.

Issuing DmiGetRowReq displays an example of how to issue the DmiGetRowReq command to the MI.

Issuing DmiGetRowReq

```
ULONG IssueGetRow(ULONG CompID,ULONG GroupID,DMI_GetRowCnf_t *Row)   // get the list group started
{
DMI_GetRowReq_t *GetRow;    // command block to use
DMI_GroupKeyData_t *NewKey, *ThisKey;
USHORT x;  // just a counter
char *Working;
DMI_STRING *Work;
DMI_UNSIGNED Size;
ULONG RC;

    GetRow = (DMI_GetRowReq_t *)malloc(4000);
    memset((void *)GetRow,0,sizeof(DMI_GetRowReq_t));
```

```
    GetRow->iGroupId = GroupID;
    GetRow->iComponentId = CompID;
    GetRow->DmiMgmtCommand.iLevelCheck = DMI_LEVEL_CHECK;
    GetRow->DmiMgmtCommand.iMgmtHandle = YOUR_MGMT_HANDLE;          // set the app handle
    GetRow->DmiMgmtCommand.iCmdHandle =  YOUR_COMMAND_HANDLE;       // point to the addinfo block
    GetRow->DmiMgmtCommand.iCnfBufLen = 8000UL;                     // set the size of the response
    GetRow->DmiMgmtCommand.pCnfBuf = (void *)malloc(8000UL);        // set up the response buffer
    GetRow->DmiMgmtCommand.iRequestCount = 1;
    GetRow->DmiMgmtCommand.iCmdLen = 4000L;
    if(Row == (DMI_GetRowCnf_t *)NULL) GetRow->DmiMgmtCommand.iCommand = DmiGetFirstRowCmd;  // set the
                                                                                             // command
    else{
        GetRow->DmiMgmtCommand.iCommand = DmiGetNextRowCmd;   // get the next row in the table
        ThisKey = (DMI_GroupKeyData_t *)((char *)Row + Row->oGroupKeyList);
        NewKey = (DMI_GroupKeyData_t *)((char *)GetRow + sizeof(DMI_GetRowReq_t));
        GetRow->oGroupKeyList = sizeof(DMI_GetRowReq_t);
        GetRow->iGroupKeyCount = Row->iGroupKeyCount;
        Working = (char *)((char *)NewKey + (Row->iGroupKeyCount * sizeof(DMI_GroupKeyData_t)));
        for(x = 0;x != Row->iGroupKeyCount;x++,ThisKey++,NewKey++){
            memcpy(NewKey,ThisKey,sizeof(DMI_GroupKeyData_t));   // move the key block over
            switch(ThisKey->iAttributeType){   // switch on the data type
                case MIF_COUNTER:
                case MIF_COUNTER64:
                case MIF_GAUGE:
                case MIF_INT:
                case MIF_INTEGER64:
                    Size = sizeof(DMI_UNSIGNED);
                    goto FinishGroup;
                case MIF_DATE:
                    Size = sizeof(DMI_TimeStamp_t);
                    goto FinishGroup;
                case MIF_DISPLAYSTRING:
                case MIF_OCTETSTRING:
                    Work = (DMI_STRING *)((char *)Row + ThisKey->oKeyValue);
                    Size = (Work->length + sizeof(DMI_UNSIGNED));
                    FinishGroup:
                    memcpy(Working,(char *)((char *)Row + ThisKey->oKeyValue),Size);
                    NewKey->oKeyValue = (DMI_OFFSET)((char *)Working - (char *)GetRow);
                    Working += Size;
                    break;
                case MIF_UNKNOWN_DATA_TYPE:    // this is the error case
                default:
                    break;
            }
        }
    }
    if((RC = DmiInvoke((DMI_MgmtCommand_t *)GetRow)) != SLERR_NO_ERROR) {   // ask for the description
        free(GetRow->DmiMgmtCommand.pCnfBuf);
        free(GetRow);
        free((char *)Handle);   // free up the add information (AddInfo) block
    }
    return RC;
}
```

DmiGetRowCnf

On return from the DmiGetRowReq call, the confirm buffer contains a DmiGetRowCnf block, followed by one or more DmiGetAttributeCnf blocks.

The format for the command block is:

Table 40. DmiGetRowCnf Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iGroupId
4	4	INT	iGroupKeyCount

```
8               4               OFFSET          oGroupKeyList
12              4               INT             iAttributeCount
16              12              STRUCT          DmiGetAttributeCnf[]
```

Variable Name	Variable Description
iGroupId	The ID of the group.
iGroupKeyCount	The number of DmiGroupKeyData blocks pointed to by oGroupKeyList.
oGroupKeyList	Offset to a list of DmiGroupKeyData blocks. iGroupKeyCount is the number of blocks in the list.
iAttributeCount	The number of DmiGetAttributeCnf blocks immediately following this block.
DmiGetAttributeCnf[]	The list of DmiGetAttributeCnf blocks.

In the case where the size of a group exceeds the size of the calling application's confirm buffer, the calling application receives a status of 1, indicating that the command was successful but that there is more of the group to be retrieved. In this case, the command can be re-issued to retrieve the rest of the group. To do this, the calling application sets the iAttributeId field to the value of the last attribute returned in the previous Get Row command. The component instrumentation then returns the attribute values that follow this attribute.

Processing DmiGetRowCnf for DmiGetFirstRowCmd and DmiGetNextRowCmd displays an example of how to handle the DmiGetRowCnf callback command block in response to the DmiGetFirstRowCmd and DmiGetNextRowCmd commands. For this example, all callbacks that are returned to the management application are processed through a single entry point.

Processing DmiGetRowCnf for DmiGetFirstRowCmd and DmiGetNextRowCmd

```
case DmiGetFirstRowCmd:    // response to the query row information
case DmiGetNextRowCmd:
    if(miCommand->iStatus == SLERR_NO_ERROR){   // we've got a good return here
        RowConfirm = (DMI_GetRowCnf_t *)miCommand->pCnfBuf;

        // Do whatever your application needs to do here

    }
    break;
```

Component Interface (CI) Command Blocks

The following command blocks are used when issuing commands to the Component Interface (CI).

- DmiCiInstallData
- DmiRegisterCiInd
- DmiRegisterCnf
- DmiUnregisterCiInd
- DmiEventData

DmiCiInstallData

To install a new MIF file, the component instrumentation fills out a DmiCiInstallData block and sends it to the service layer with the DmiInvoke() function call.

The format for the command block is:

Table 41. DmiCiInstallData Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId
68	4	INT	iFileCount
72	8	STRUCT	DmiFileData[ ]

Variable Name	Variable Description
DmiMgmtCommand	The command block. The value of iCommand is 0x402. iMgmtHandle should always be set to zero.
iComponentId	The service-layer defined ID of this component. On initial install, this is set to zero. On subsequent installs, this field is set to the component's ID. In this case, the field is used to provide additional MIF data for this component, such as MIF files for additional languages.
iFileCount	Number of DmiFileData structures that follow.
DmiFileData[ ]	A list of structures, as defined in DmiFileData Command Block.

The format of the DmiFileData block is:

Table 42. DmiFileData Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iFileType
4	4	OFFSET	osFileData

Variable Name	Variable Description
iFileType	Type for this component's MIF files. The type code is a 32-bit values. If the value is even, the contents of osFileData indicates a file name. If the value is odd, the contents of osFileData indicates the MIF data in memory. The following type codes are defined: **Type code** — **Meaning** 0 — Reserved, do not use 1 — Reserved, do not use 2 — MIF file name 3 — MIF file pointer 4 — SNMP mapping file name 5 — SNMP mapping file pointer 0x80000000 — Start of implementation-specific types Note that like command codes, file type codes above x80000000 are reserved for implementation-specific use.
osFileData	Offset to a string whose contents depend on the type code.

Issuing DmiCiInstallData displays an example of how to issue the DmiCiInstallData command to the CI.

Issuing DmiCiInstallData

```
DmiLibInstallData_t *InstallMIF(char *MIF_FileName)   // Receive thread -- handles all confirm processing
{
DMI_CiInstallData_t *inst;
```

```
ULONG               reqSize,FileOffset,FileSize;
DMI_STRING          *FileName;
ULONG               RC = SLERR_OUT_OF_MEMORY;

    FileName = (DMI_STRING *)malloc((ULONG) (strlen(MIF_FileName) + sizeof(ULONG)));
    if(FileName != (DMI_STRING *)NULL){  // we got the memory we asked for
        memcpy(FileName->body,MIF_FileName,(FileName->length = strlen(MIF_FileName)));  // set up the file
                                                                                        // name string
        FileOffset = reqSize = (ULONG)sizeof(DMI_CiInstallData_t);   // install structure includes ONE
                                                                      // file structure
        reqSize += (ULONG) (FileName->length + sizeof(FileName->length));   // add length of DMI string
        inst = (DMI_CiInstallData_t *)malloc(reqSize);
        if(inst != (DMI_CiInstallData_t *)NULL){   // we've got the memory
            memset(inst,0,reqSize);   /* clear out the whole thing */
            inst->DmiMgmtCommand.iLevelCheck = DMI_LEVEL_CHECK;
            inst->DmiMgmtCommand.iCmdLen = reqSize;
            inst->DmiMgmtCommand.iCmdHandle = 1;
            inst->DmiMgmtCommand.iRequestCount = 1;
            inst->DmiMgmtCommand.iCommand = DmiCiInstallCmd;
            inst->iFileCount = 1;
            inst->DmiFileListÝ0¨.iFileType = MIF_MIF_FILE_NAME_FILE_TYPE;
            inst->DmiFileListÝ0¨.oFileData = FileOffset;
            FileSize = (size_t) (sizeof(FileName->length) + FileName->length);
            memcpy(((char *)inst + FileOffset),FileName, FileSize);
            RC = DmiInvoke((DMI_MgmtCommand_t *)inst);
            if(RC != SLERR_NO_ERROR) free(inst);
        }
        free(FileName);
    }
    return RC;
}
```

DmiCiUninstallData

To remove a component from the MIF database, the component instrumentation fills out a DmiCiUninstallData block and sends it to the service layer with the DmiInvoke() function call.

The format for the command block is:

Table 43. DmiCiUninstallData Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId

Variable Name	Variable Description
DmiMgmtCommand	The command block. The value of iCommand is 0x403. iMgmtHandle should always be set to zero.
iComponentId	The service-layer defined ID of this component.

Issuing DmiCiUninstallData displays an example of how to issue the DmiCiUninstallData command to the CI.

Issuing DmiCiUninstallData

```
ULONG UnInstallMIF(ULONG CompID)   // Receive thread -- handles all confirm processing
{
DMI_CiUninstallData_t Unload;
ULONG RC;

    memset(&Unload,0,sizeof(DMI_CiUninstallData_t));   // clear it out first
    Unload.DmiMgmtCommand.iLevelCheck = DMI_LEVEL_CHECK;
```

```
    Unload.DmiMgmtCommand.iCommand = DmiCiUninstallCmd;
    Unload.DmiMgmtCommand.iCmdLen = sizeof(DMI_CiUninstallData_t);
    Unload.DmiMgmtCommand.iMgmtHandle = YOUR_MGMT_HANDLE;
    Unload.DmiMgmtCommand.iCmdHandle  = YOUR_COMMAND_HANDLE;
    Unload.DmiMgmtCommand.iRequestCount = 1;
    Unload.iComponentId = CompID;
    return DmiInvoke((DMI_MgmtCommand_t *)&Unload);    // issue the unload command
}
```

DmiRegisterCiInd

The DmiRegisterCiInd block can be used as follows:

- To register a callable interface for components that have resident instrumentation code
- To get the version of the service layer.

The format for the command block is:

Table 44. DmiRegisterCiInd Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	64	STRUCT	DmiMgmtCommand
68	4	INT	iComponentId
72	4	PTR	pAccessFunc
76	4	PTR	pCancelFunc
80	4	INT	iAccessListCount
84	8	STRUCT	DmiAccessData[ ]

Variable Name	Variable Description
DmiMgmtCommand	The command block. The value of iCommand is 0x400.
iComponentId	ID of the service layer assigned to the component on installation.
pAccessFunc	Pointer to the entry point of the component instrumentation. The service layer calls the component instrumentation with the C prototype: `unsigned long  pAccessFunc(PTR command)` The variable *command* is the complete block. A value of zero is illegal.
pCancelFunc	Pointer to the entry point that the service layer calls to cancel an outstanding operation through the direct interface. The service layer calls with the C prototype: `unsigned long  pCancelFunc(void)` A value of zero is illegal.
iAccessListCount	The number of blocks of type DmiAccessData, as defined in DmiAccessData Command Block.
DmiAccessData[ ]	The groups and individual attributes that use the direct interface. The format of the DmiAccessData block is defined in DmiAccessData Command Block.

The format of the DmiAccessData block is:

Table 45. DmiAccessData Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iGroupId
4	4	INT	iAttributeId

Variable Name	Variable Description
iGroupId	Group that uses the direct interface. A value of zero indicates that all groups within this MIF file use the direct interface. In this case, the iAttributeId field is ignored.
iAttributeId	Attributes, within the group specified by iGroupId, that use the direct interface. A value of zero indicates that all attributes within this group use the direct interface.

Issuing DmiRegisterCiInd displays an example of how to issue the DmiRegisterCiInd command to the CI.

Issuing DmiRegisterCiInd

```
ULONG RegisterCI(ULONG ComponentID)
{
ULONG Size,x;
DMI_RegisterCiInd_t    *ciRegister;
ULONG RC = SLERR_OUT_OF_MEMORY;
DMI_MgmtCommand_t      *dmiCommand;
DMI_AccessData_t       *accessList;


   Size = (ULONG) (sizeof(DMI_RegisterCiInd_t) + (3 * sizeof(DMI_AccessData_t)));   // get size of block
   ciRegister = malloc(Size);
   if(ciRegister != (DMI_RegisterCiInd_t *)NULL){
       memset(ciRegister,0,Size);   // clear out the whole thing first
       dmiCommand = &(ciRegister->DmiMgmtCommand);
       dmiCommand->iLevelCheck = DMI_LEVEL_CHECK;
       dmiCommand->iCommand = DmiRegisterCiCmd;
       dmiCommand->iCmdLen = Size;
       dmiCommand->iCnfBufLen = 4000UL;
       dmiCommand->pCnfBuf = malloc(4000UL);

       ciRegister->iComponentId = ComponentID;          // assign the ID from install time
       ciRegister->pAccessFunc = DmiCiInvoke;           // invoke entry point into component code
       ciRegister->pCancelFunc = DmiCiCancel;           // Cancel entry point into component code
       ciRegister->iAccessListCount = 4;                // this example has four attributes in one group
       accessList = &(ciRegister->DmiAccessListÝ0¨);
       for(x = 1;x != 5;x++){
           accessList->iGroupId = 2;                    // assign the group ID from the MIF
           accessList->iAttributeId = x;                // assign the attribute DI from the MIF
           accessList++;
       }
       RC = DmiInvoke((DMI_MgmtCommand_t *)ciRegister);
       free(ciRegister);
    }
    return RC;
}
```

DmiRegisterCnf

On return from the DmiRegisterCiInd call, the service layer places a DmiRegisterCnf block in the response buffer. This block is the same as that described in DmiRegisterCnf Command Block.

The format for the command block is:

Table 46. DmiRegisterCnf Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	8	STRUCT	DmiVersion
8	4	INT	iDmiHandle

Variable Name	Variable Description
DmiVersion	A block defining the version of the service layer.
iDmiHandle	Set to a unique handle for this component instrumentation. This is only valid for as long as the component instrumentation is registered with the service layer. The handle is passed to the service layer when unregistering.

A component can register more than once. This is necessary when a component has multiple entry points for instrumentation. In this case, the component instrumentation is assigned multiple handles.

Processing DmiRegisterCnf displays an example of how to handle the DmiRegisterCnf callback command block. For this example, all callbacks that are returned to the management application are processed through a single entry point.

DmiUnregisterCiInd

DmiUnregisterCiCmd instructs the service layer to remove a direct component instrumentation interface from the service layer's table of registered interfaces. The call uses the DmiUnregisterCiInd block.

The format for the command block is:

Table 47. DmiUnregisterCiInd Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iCiHandle

Variable Name	Variable Description
DmiMgmtCommand	The command block. The value of iCommand is 0x401.
iCiHandle	The handle that the service layer assigned to the component on registration. Components must unregister once for each registration performed. No return buffer is required or used.

Issuing DmiUnregisterCiInd displays an example of how to issue the DmiUnregisterCiInd command to the CI.

Issuing DmiUnregisterCiInd

```
ULONG UnRegisterCI(ULONG iCiHandle)
{
DMI_UnRegisterCiInd_t   *ciUnRegister;
ULONG                   RC = SLERR_OUT_OF_MEMORY;
DMI_MgmtCommand_t       *dmiCommand;

   ciUnRegister = malloc(sizeof(DMI_UnRegisterCiInd_t));
   if(ciUnRegister != (DMI_UnRegisterCiInd_t *)NULL){
       memset(ciUnRegister,0,sizeof(DMI_UnRegisterCiInd_t));   // clear out the whole thing first
       dmiCommand = &(ciUnRegister->DmiMgmtCommand);
       dmiCommand->iLevelCheck = DMI_LEVEL_CHECK;
       dmiCommand->iCommand = DmiUnregisterMgmtCmd;
```

```
        dmiCommand->iCmdLen = sizeof(DMI_UnRegisterCiInd_t);
        ciUnRegister->iCiHandle = iCiHandle;          // assign the ID from install time
        RC = DmiInvoke((DMI_MgmtCommand_t *)ciUnRegister);
        free(ciUnRegister);
    }
    return RC;
}
```

DmiEventData

To send an event, the component instrumentation sends a DmiIndicate block to the service layer's DmiIndicate() entry point. The event data is defined in a DmiEventData block that corresponds to the oIndicationData field in the DmiIndicate block.

The format for the command block is:

Table 48. DmiEventData Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iClassCount
4	4	STRUCT	DmiClassData[ ]

Variable Name	Variable Description
iClassCount	The number of group class strings in the component.
DmiClassData[ ]	A list of the group class data blocks for this component, as defined in DmiClassData Command Block.

The format of the DmiClassData block is:

Table 49. DmiClassData Command Block

OFFSET	SIZE	TYPE	VARIABLE NAME
0	4	INT	iComponentId
4	4	OFFSET	osClassString
8	4	INT	iRowCount
12	4	STRUCT	DmiGetRowCnf[ ]

Variable Name	Variable Description
iComponentId	The ID of the component that this event pertains to.
osClassString	Offset to the string that names the group class for the event.
iRowCount	Number of DmiRowList structures below.
DmiGetRowCnf[ ]	A list of one or more DmiGetRowCnf structures that describes the data in the event.

Enabling a Product for the DMI

This chapter describes how to enable a product to be managed as a component through the DMI.

The general tasks to enable a product include:

- Defining which aspects of the product are to be managed
- Writing a Management Information Format (MIF) file to describe the manageable features
- Writing instrumentation code to provide the service layer with information about the component
- Determining what errors, exceptions, or problems are to be forwarded to the service layer as indications
- Installing the MIF into the MIF database

Defining the Product's MIF File

A MIF file is an ASCII text file that conforms to the Management Information Format (MIF) described in the *DMI Specification* . The MIF file for a component defines the manageable aspects of the component and provides this information to management applications through the DMI. Because the DMI does not assume that a management application can determine anything about a component in the system, it is important that the component provide all significant information about its manageable characteristics through the MIF file.

The steps involved for defining a MIF file include:

- Determining what characteristics of the component can be managed
- Defining the groups into which these characteristics will be organized
- Defining the attributes that make up the groups

A MIF file can contain as few as one group with the six standard attributes or as many groups and attributes as you choose.

Determining Attributes

Before you begin determining possible grouping arrangements for the attributes in your component, be sure that you have considered all of the manageable aspects of your component in as much detail as possible. Be sure to include all of the items that can be configured or that can be changed, as well as information that enables asset management, such as a product version number or a serial number.

The ComponentID group (ComponentID Group) is a good example of these kinds of attributes.

You can also examine existing MIF files of other components to get an idea of what kinds of attributes are defined to enable a product for management through the DMI.

Defining Groups

After you have determined all of the attributes that apply to your component, you must define the groups into which these attributes will be organized. When you are defining the groups and attributes for the component, consider how the component will be used. The groups you define should be conceptually distinct from each other and should pertain to a manageable aspect of your component, whether that is a physical or logical element of the component.

If your component is a software product, you might want to define your groups according to the functions a user can perform with the product. A word-processing program, for example, might require groups that describe the editing function, the spell-checker, the drawing tools, and the printing function.

If your component is a hardware product, such as a LAN adapter card, you might want to relate your groups more closely to the physical aspects of the device. In the case of a LAN adapter card, the defined groups could describe such things as the port adapter statistics, the adapter hardware, and the operational state of the adapter.

Keep the following points in mind when defining your component's groups:

- Whenever possible, use the standard MIF files and groups specified by the DMTF, instead of creating private, non-standard groups. Using these common groups allows more management applications to manage your component. The standard groups that are available are defined in the following standard MIF files maintained by the DMTF:

  - Network Interface Card (NIC) MIF
  - Desktop System MIF
  - Large Mailroom Operation (LMO) MIF
  - Printer MIF
  - Software MIF

  Note: When you use a standard group, change only the attribute value for any attributes in the group. For those attributes that you do not use, specify the Unsupported keyword for the value. Do not change any other items in the standard attributes, such as attribute ID or attribute type, and do not renumber the attributes within a standard group.

- Consult existing proprietary MIF files or groups for ideas about how to model difficult aspects of your component. You might find that another component's MIF file has addressed a similar problem.

- Keep the number of attributes in a group to 20 or fewer, if possible. Small groups are easier to manage, and their modular design makes them easier for you to reuse than a few large groups. Even only one or two attributes in a group is reasonable, as long as the grouping is still logical.

- Define your groups to be reusable. Although you are creating a MIF file for a single component, you might be able to use the defined groups in MIF files for other components you design.

- Allow for different views of the same data by providing both standard public groups of attributes and private, individual groups of attributes. For instance, both the external fax-modem and the internal built-in modem can be described by the modem group, but each would need an individual group to describe characteristics that do not apply in the standard group.

Defining Attributes

After you determine the manageable aspects of your component and define the groups you will use to organize the attributes, you are ready to define the attributes according to the MIF. Understanding the MIF provides a detailed description of the syntax and conventions used by the MIF, including the different definitions used to define your MIF file.

The attribute is the smallest piece of your component that can be defined and identifies a discrete manageable aspect of the component. The way you define an attribute affects not only how the attribute value is stored and accessed but also how the component is managed. To ensure that management of your component is as efficient as possible, consider these points when defining the attributes in your component:

- Access to an attribute should be determined by the likelihood of its changing. If it is clear that an attribute value will not change or should not be modified, define the attribute with read-only access.

  For those attributes that do change, determine whether the value should be available to a management application. If so, read-write access is appropriate for the attribute, but if not, specify write-only access for the attribute.

- Tables provide flexibility for defining and managing attributes that require more than one value. As your component develops, the tables you have defined can be expanded to accommodate any changes.

  When defining key attributes to be used as indexes into a table, use as many keys as necessary to provide management applications efficient and flexible access to the data in the table. The attributes you specify as keys should be appropriate to the component and not simply arbitrary indexing values.

  If you are developing a component that does not have instrumentation associated with it, ensure that the order of the keys in the MIF file is sequential. This maximizes performance of the service layer, particularly across SNMP.

  Note: The values for each row in a table, including the key value, must be supplied either from the MIF database or by component instrumentation. You cannot supply values from the database and from instrumentation for the same row.

- Do not add attributes specific to your component to those defined for a standard group. If you alter the contents of a standard group, management applications can no longer manage the group reliably. Instead, use the standard group in your component and add another group that contains your private attribute definitions.

- Use detailed description statements to provide as much information about your attributes as possible. Information such as

minimum, maximum, and default values is also useful.

- Designate how an attribute value is to be supplied according to the stability and nature of the attribute. If the attribute value is static, the value can be supplied by the MIF database. However, if the attribute value is likely to change frequently, specify that the value be supplied by component instrumentation, such as a runtime program or a direct-interface program. This ensures that the value provided for an attribute is current.

  For example, ProductID and SerialNumber are likely candidates for database attributes because their values are not likely to change. However, Available_Bytes_on_Hard_Drive is an attribute that is appropriate for component instrumentation.

Writing the MIF File

You can write the MIF file with any editor you choose, as long as it is a plain ASCII text file with no embedded formatting commands or control characters. Although the order of group definitions in the MIF file does not affect the way the file is interpreted by the service layer, the structure of the MIF file should be logical and organized for easy maintenance and expansion. Use white space, indentation, and comments to enhance the readability of the file.

The conventions and syntax used by the Management Information Format are described in detail in Understanding the MIF.

Follow these general steps when writing the MIF file:

- Define your groups and specify which are standard and which are private to the component. Use the ComponentID group as the first group in the file.

  If a group functions as a table template, specify a key value and an ID, if appropriate.

- Define any enumeration definitions that apply to the component as a whole.

- Define the attributes for each group.

- Define any table definitions that refer to previously defined templates, if appropriate.

Writing Instrumentation Code

When a component receives a request from the service layer for an attribute's value, the instrumentation code for the component is invoked to retrieve the value. By providing instrumentation for your component, you ensure that the information that is accessed by management applications is both current and accurate. Without instrumentation, you run the risk of allowing the information in the MIF file to become outdated.

You can provide instrumentation for your component in one of two ways:

- An overlay program
- A direct-interface program

Overlay Programs

An overlay program is invoked by the service layer whenever a management application requests to retrieve or set the value of an attribute in the component. The overlay program is loaded into the service layer's process space and enables the service layer to directly access the attribute values through the overlay program's procedures. When the request is completed, the service layer unloads the overlay program.

Overlay programs are useful for components that do not already have resident code or for components running in systems with constrained memory. They are also appropriate if they are called infrequently. The data associated with your component is not retained by the overlay program between invocations, unless you specifically designate some storage mechanism for the values. If your component data must always be available, a direct-interface program is more appropriate.

Direct-interface Programs

A direct-interface program runs continuously as a separate process in the system. As part of its initialization, the program registers with the service layer and indicates the component with which the program is associated. When the service layer receives requests from management applications for information in that component, the service layer passes the request over to the direct-interface program, which returns the appropriate values.

An advantage of the direct-interface program is its ability to maintain up-to-date component information. Because the program is always running, the attribute values for the component are always current and are not discarded between requests. The code for a direct-interface program can be added to the device driver code for the component (if appropriate) or to any other program that is loaded in the background.

A disadvantage of the direct-interface program is that it is always loaded and utilizing resources.

Coding Considerations

Instrumentation code, whether for an overlay program or a direct-interface program, must be designed to process the following DMI commands:

- DmiGetAttributeCmd
- DmiSetAttributeCmd
- DmiSetReserveAttributeCmd
- DmiSetReleaseAttributeCmd
- DmiGetRowCmd
- DmiGetFirstRowCmd
- DmiGetNextRowCmd

These commands are all accessed through the DmiCiInvoke function call, which is sent from the service layer. In addition to these commands, the instrumentation code must also issue the pConfirmFunc function call for every command, including set commands and commands that generate errors.

For the DmiGetFirstRowCmd and DmiGetNextRowCmd commands, the DmiGetRowCnf structure is generated by the service layer before it issues the DmiCiInvoke function call. The GroupKeyData structures are also allocated, and the oGroupKeyList variable in the DmiGetRowCnf structure is supplied. Your instrumentation code can only modify the confirm buffer and not the DmiInvoke command buffer.

For a complete description of the commands and structures used by the DMI, read DMI Command Blocks

The SystemView Agent program also includes the DMI procedures library (DMIAPI) to simplify the writing of instrumentation code. By using the procedures provided by the DMI API and by patterning your code after the sample programs included in the SystemView Agent, many considerations regarding function calls and DMI commands are resolved for you. For a detailed explanation of the DMI procedures library, read DMI Procedure Library (DMIAPI).

Overlay Programs

To enable the service layer to properly access the overlay program for your component, ensure that you do the following when writing your MIF file:

- Specify the name of the component instrumentation as part of the path definition for your component. For example:

```
start paths
      name = "EXPOverlay"
      os2  = "names"
      unix = "/usr/lpp/sva/bin/names"
end paths
```

  In this case, the value of the name statement is used in attribute definitions to refer back to this path definition. The operating system statement identifies the filename of an OS/2 dynamic link library (DLL) that functions as the overlay program.

- In the definition for any attribute that requires instrumentation to provide its value, specify the path definition. For example, in the attribute definition use the following statement:

  ```
  value = * "EXPOverlay"
  ```

  This identifies the previous path definition by its name statement and enables the service layer to load the proper overlay program to retrieve the attribute's value.

For implementation details specific to the operating environments supported by SystemView Agent:

Read	For information about
Implementing DMI on OS/2	OS/2
Implementing DMI on AIX.	AIX
Implementing DMI on Windows NT/Windows 95.	Windows NT and Windows 95

Direct-interface programs

To use a direct-interface program to retrieve attribute values, your component must register with the service layer as a direct-interface component. When it registers, the component passes its entry point to the service layer. This enables the service layer to pass control to the direct-interface program when an attribute's value is required. When the direct-interface program unloads, it unregisters with the service layer.

You can indicate that your component relies on a direct-interface program by using the **direct-interface** keyword in the path definition of the component's MIF file. For example:

```
start paths
      name = "NAMES - Direct Interface"
      os2 = direct-interface
      unix = direct-interface
end paths
```

This ensures that the service layer can return an appropriate error if it attempts to access the value for an attribute and the direct-interface program is not running.

For implementation details specific to the operating environments supported by SystemView Agent:

Read	For information about
Implementing DMI on OS/2	OS/2
Implementing DMI on AIX.	AIX
Implementing DMI on Windows NT/Windows 95.	Windows NT and Windows 95

Sending Events

When a component encounters an error situation or needs to notify a management application of a status change, for example, the component sends an event to the service layer. If a management application has requested to receive unsolicited notifications from that component, the service layer forwards the event to the management application as an indication.

You can use events to enhance the manageability of your component by providing detailed information about the status of the component and by notifying management applications of these changes in a timely manner. Events are especially useful when a serious error or similar change has occurred and the management application must be informed immediately.

Events are issued with the DmiIndicate function call. The service layer processes the events as they are received and returns a status value indicating the success or failure of the DmiIndicate call. For more information about the DmiIndicate call and its associated command blocks, read DmiIndicate Command Block.

Installing Components

When you install your component on a system, the installation procedure either incorporates the MIF file for the component into the MIF database or, if the service layer is not running, installs the file in the MIF subdirectory. To accomplish this, you can use the DmiInstall procedure that is included in the DMI procedure library. This procedure handles such tasks as adding the MIF file to the MIF database and determining whether the service layer is running.

For a detailed explanation of the DMI procedures library, read DMI Procedure Library (DMIAPI).

For implementation details specific to the operating environments supported by SystemView Agent:

Read	For information about
Implementing DMI on OS/2	OS/2
Implementing DMI on AIX.	AIX
Implementing DMI on Windows NT/Windows 95.	Windows NT and Windows 95

Uninstalling Components

The uninstall procedure for your component should do the following:

- If the component uses a direct-interface program, ensure that the program issue a DmiCiUnregister command, if the program is registered to the service layer and currently running,
- Remove the MIF file for the component from the MIF database by issuing a DmiCiUninstall command.

  To issue this command, your uninstall procedure must identify which component ID in the MIF database corresponds to your component.

DMI Procedure Library (DMIAPI)

The DMI procedure library provided with the SystemView Agent package is a C library that contains procedures for a specific operation system for developing applications that use the DMI Management Interface and Component Interface entry points. You can use the procedures in the library to simplify the processes of installing components in the MIF database and invoking the service layer's Component Interface. The procedures in the DMI procedure library are used in the example programs provided with the SystemView Agent program.

Library Directories

This tables in this section display the subdirectories in which the files in the DMI procedure library are located, according to the operating system:

Operating system	Table
OS/2	DMI Procedure Library Directories for OS/2
AIX	DMI Procedure Library Directories for AIX

Table 50. DMI Procedure Library Directories for OS/2

FILE	DIRECTORY
DMIAPI.H	"[BOOT DRIVE]:"\INCLUDE
DMI.H	"[BOOT DRIVE]:"\INCLUDE
ERROR.H	"[BOOT DRIVE]:"\INCLUDE
OS_DMI.H	"[BOOT DRIVE]:"\INCLUDE
DMIAPI.LIB	"[BOOT DRIVE]:"\LIB
DMIAPI.DLL	"[BOOT DRIVE]:"\BIN

NOTE: The DMI.H, ERROR.H, and OS_DMI.H files are all included by DMIAPI.H. The only header file a developer needs to include is DMIAPI.H.

Table 51. DMI Procedure Library Directories for AIX

FILE	DIRECTORY
dmiapi.h	/usr/lpp/sva/include
dmi.h	/usr/lpp/sva/include
error.h	/usr/lpp/sva/include
os_dmi.h	/usr/lpp/sva/include
libdmisl.a	/usr/lpp/sva/lib
libdmiapi.a	/usr/lpp/sva/lib
(for applications that do not use X-windows)	
libdmiapiX.a	/usr/lpp/sva/lib
(for applications that use X-windows)	

NOTE: dmi.h, error.h, and os_dmi.h are all included by dmiapi.h. The only header file a developer needs to include is dmiapi.h.

Table 52. DMI Procedure Library Directories for Windows NT/Windows 95

FILE	DIRECTORY
DMI_API.H	"C:\SVA"\DMI\INCLUDE
DMI.H	"C:\SVA"\DMI\INCLUDE
ERROR.H	"C:\SVA"\DMI\INCLUDE

OS_DMI.H	"C:\SVA"\DMI\INCLUDE
DMIAPI32.LIB	"C:\SVA"\DMI\LIB
DMIAPI.DLL	"C:\SVA"\DMI\LIB

NOTES:

- "C:\SVA" indicates the default drive and directory where SystemView Agent is installed. If you installed the product in a different path, these file locations will differ accordingly.
- The DMI.H, ERROR.H, and OS_DMI.H files are all included by DMI_API.H. The only header file a developer needs to include is DMI_API.H.

Procedures and Data Types

This section describes the procedures and data types that make up the DMI procedure library. The procedures include:

- DmiCiProcess()
- DmiInstall()
- DmiInvoke()
- DmiIndicate()

DmiCiProcess()

This procedure is called by DmiCiInvoke(). It processes a command through the service layer's Component Interface.

Syntax

```
DMI_UNSIGNED DMI_FUNC_ENTRY DmiCiProcess (DmiCiControl_t *Command,
            void _FAR*dmiCommand);
```

Parameters

Command	A pointer to the current Component Interface command block.
dmiCommand	A pointer to the current command block passed by the service layer to the program through DmiCiInvoke.

DmiInstall()

This procedure allows an application developer writing a product setup or installation program to easily add MIF files to the MIF database.

Syntax

```
DmiLibInstallData_t *DMI_FUNC_ENTRYDmiInstall(   int iFileCount,
               DmiLibFileData_t *dmiLibFileList,
               DMI_STRING *pDmiDir,
               int iDefineDmiDir
               int StatusCall);
```

Parameters

iFileCount	The number of files within the dmiFileList structure.
dmiLibFileList	An array of structures containing the types and locations of the files to install.
pDmiDir	A DMI_STRING pointer to an operating system definition for the DMIDIR environment variable. If this field is NIL, the library uses the system definition. When this field is specified, the library uses this path name in place of the system definition. This parameter is ignored by the OS/2 implementation.
iDefineDmiDir	A Boolean value (for which non-zero is true) that directs the library to modify the system startup file to include a definition for the DMIDIR environment variable. If pDmiDir is not NIL, DMIAPI.LIB uses the value of that field in the modified system startup file. If pDmiDir is NIL, DMIAPI.LIB uses the current value of the DMIDIR environment variable. If DMIDIR is not defined, DMIAPI.LIB attempts to query the service layer for the path name of the DMI directory. If DMIAPI.LIB is unable to determine the path name of the DMI directory, DmiInstall() terminates with an error. To resolve this condition, the calling program must re-issue the DmiInstall() command, with the pDmiDir parameter defined. If the system startup file already had a definition for DMIDIR, then that definition will be removed, and the new definition will be added. This parameter is ignored by the OS/2 implementation.
StatusCall	A callback entry point that the service layer calls with the MIF compiler messages. The Component Installation Status window in the DMI browser is an example of how you can use this parameter to display messages.

DmiInvoke()

The DMI Specification describes the MI entry point of the service layer. The entry point is called DmiInvoke(). The DmiInvoke() procedure is used by the calling application to issue commands to the service layer.

Syntax

```
DMI_UNSIGNED DMI_FUNC_ENTRY DmiInvoke(void _FAR *dmiMgmtCommand);
```

Parameters

dmiMgmtCommand	A pointer to a DMI command

Notes

If the service layer is not running, an SLERR_SL_INACTIVE error is returned. Otherwise the return value from DmiInvoke() is the value returned by the service layer when the command was issued. Error codes are documented in Status Codes

---

DmiIndicate()

This procedure operates like DmiInvoke but is the indication entry point of the service layer.

Syntax

```
DMI_UNSIGNED DMI_FUNC_ENTRY DmiIndicate(DMI_Indicate_t _FAR
                                        *dmiIndication);
```

Parameters

dmiIndication	A pointer to a DMI indication block.

Notes

If the service layer is not running, an SLERR_SL_INACTIVE error is returned.

Data Types

The data types used by the procedures in the DMI procedure library are described in this section. For each data type, the syntax is given and the parameters are explained.

DMI_STRING

The syntax for using DMI_STRING is:

```
typedef struct {
        unsigned long length;
        char body[1];
} DMI_STRING;
```

Parameter	Description
length	The length of the string
body	The data for the string. The string is not null-terminated. The 1 in the definition is strictly for ANSI C compliance. The actual size will be the size of the string.

DmiLibBoolean

The syntax for using DmiLibBoolean is:

```
typedef enum {DmiLibFalse, DmiLibTrue} DmiLibBoolean_t;

DmiLibFalse  0
DmiLibTrue   1
```

DmiLibFileType

The syntax for using DmiLibFileType is:

```
typedef enum {MifFileName = 2, MifFilePointer} DmiLibFileType_t;

MifFileName      2
MifFilePointer   3
```

DmiLibFileData

The syntax for using DmiLibFileData is:

```
typedef struct {
        unsigned long iFileType;
        DMI_STRING *pFileData;
} DmiLibFileData_t;
```

Parameter	Description
iFileType	The type of the referenced file: 1 MIF file name 2 MIF file pointer
pFileData	A DMI_STRING pointer to the file data. For type 2, this pointer references the operating system path name to the file. For type 3, the pointer references a DMI_STRING object that contains the actual file data.

DmiLibInstallData

A pointer to this structure is used as the return value from calling DmiInstall(). The syntax for using DmiLibInstallData is:

```
typedef struct {
        unsigned long iComponentId;
        unsigned long iDmiLibStatus;
        unsigned long iSlStatus;
} DmiLibInstallData_t;
```

Parameter	Description
iComponentId	The component ID that the service layer assigned to the newly installed MIF file.
iDmiLibStatus	The status of the DmiInstall() procedure. Possible status codes include: DmiLibDirInstallNoError DmiLibSlInstallNoError DmiLibIllegalFileType DmiLibCannotCloseDestinationFile

DmiLibCannotCreateDestinationFile

DmiLibCannotCreateDirectory

DmiLibCannotOpenSourceFile

DmiLibCannotReadSourceFile

DmiLibCannotWriteDestinationFile

DmiLibCannotExecuteInstallCommand

DmiLibCannotRunServiceLayer

DmiLibCannotAllocateMemory

DmiLibCannotSetDmiDirEnvironmentVariable

DmiLibDmiDirEnvironmentVariableNotDefined

iSlStatus	This field is the value of the iStatus field from the result of the DmiCiInstallCmd.

DmiCiControl

The syntax for using DmiCiControl is:

```
typedef struct {
   union {
      struct DmiMgmtCommand         dmiMgmtCommand;
      struct DmiGetAttributeReq     dmiGetAttributeReq;
      struct DmiSetAttributeReq     dmiSetAttributeReq;
      struct DmiGetRowReq           dmiGetRowReq;
      char                          requestBuffer[1];
   } _FAR                           *pRequestBuffer;
   union {
        struct DmiGetAttributeCnf   dmiGetAttributeCnf[1];
        struct DmiGetRowCnf         dmiGetRowCnf;
        char                        confirmBuffer[1];
   } _FAR                *pConfirmBuffer;
   DMI_FUNC3_OUT          DmiConfirmFunc;
   enum CiBoolean         ciCancelFlag;
   struct DmiConfirm      dmiConfirm;
   DmiCiAttribute_t _FAR *ciKeyList;
  DMI_UNSIGNED                MaxKeyCount;
    CI_FUNC_IN1           CiGetAttribute;
    CI_FUNC_IN1           CiGetNextAttribute;
    CI_FUNC_IN2           CiReleaseAttribute;
    CI_FUNC_IN2           CiReserveAttribute;
    CI_FUNC_IN2           CiSetAttribute;
} DmiCiControl_t
```

Parameter	**Description**
pRequestBuffer	A pointer to the request buffer for this Component Interface command.
pConfirmBuffer	A pointer to the confirm buffer for this Component Interface command
DmiConfirmFunc	The entry point into the service layer for this command's confirm function
ciCancelFlag	A flag set when the service layer requests that this command be canceled
dmiConfirm	The confirm data block used for this command

These two parameters must be set by the calling program:

ciKeyList	A pointer to the key list array to
MaxKeyCount	The maximum number of keys for in this component, determined by the group in your MIF file which has the longest key list

---

Implementing DMI on OS/2

This chapter describes the OS/2 files provided with the SystemView Agent program for use in creating applications and component instrumentation in an OS/2 environment. The following types of files are included:

- Program files related to the service layer

- Example files for an OS/2 management application and component, including overlay and direct-interface instrumentation for the component

The chapter also presents any implementation considerations that are specific to OS/2.

Program Files

OS/2 Program Files displays a list of OS/2 program files provided with the SystemView Agent program. The paths listed here indicate the default drive and subdirectory used during installation. If you installed SystemView Agent in a different drive or directory, adjust the file location accordingly.

Table 53. OS/2 Program Files

FILE NAME	LOCATION	DESCRIPTION
DMISL.EXE	[boot drive]:\OS2	The service layer executable file. You must start this program before any applications can access the service layer.
DMISL.MSG	[boot drive]:\DMISL\BIN	The message file containing all messages associated with the service layer.
DMIAPI.DLL	[boot drive]:\OS2\DLL	The dynamic link library (DLL) for the DMI procedure library. The SystemView Agent installation process adds this subdirectory to the LIBPATH statement in your CONFIG.SYS file.
DMIPM.EXE	[boot drive]:\OS2	The executable file for the DMI MIF browser.
DMI.H	[boot drive]:\DMISL\INCLUDE	The header file for the service layer.
DMIAPI.H	[boot drive]:\DMISL\INCLUDE	The header file for the DMI procedure library. Refer to DMI Procedure Library (DMIAPI) for more information about the DMI procedure library.
OS_DMI	[boot drive]:\DMISL\INCLUDE	The header file for the OS/2 operating system.
ERROR.H	[boot drive]:\DMISL\INCLUDE	The header file for the service layer's error message file (DMISL.MSG).
DMIAPI.LIB	[boot drive]:\DMISL\LIB	The DMI import library for use with version 2 of the IBM C/C++ Tools product. Refer to DMI Procedure Library (DMIAPI) for more information about the DMI procedure library.

Example Files

OS/2 Example Files displays a list of OS/2 example files provided with SystemView Agent. The paths listed here indicate the default drive and subdirectory used during installation. If you installed SystemView Agent in a different drive or directory, adjust the file location accordingly.

Table 54. OS/2 Example Files

FILE NAME	LOCATION	DESCRIPTION
MIEXAMP.C	[boot drive]:\DMISL\EXAMPLES\MI	The source code used by the DMI MIF browser to interact with the Management Interface.
NAMES.C	[boot drive]:\DMISL\EXAMPLES\CI	The source file for the overlay instrumentation for the sample component.
NAMES.DEF	[boot drive]:\DMISL\EXAMPLES\CI	The module definition file for use in building the DLL file for the overlay instrumentation. The NAMES.DEF file defines the entry points and other characteristics of the DLL.
NAMES.DEP	[boot drive]:\DMISL\EXAMPLES\CI	The dependency file for use in compiling the NAMES.C file. You might need to modify this file to accommodate path differences in your system.
NAMES.MAK	[boot drive]:\DMISL\EXAMPLES\CI	The make file for use in compiling and linking the sample component instrumentation (NAMES.C). You can use this make file with the NMAKE utility provided by version 2 of the IBM C/C++ Tools product. You might need to modify this file to accommodate path differences in your system.
NAMES.MIF	[boot drive]:\DMISL\EXAMPLES\CI	The sample MIF file that defines the NAMES component. You might need to modify the path definition in the file to accommodate path differences in your system.
NAMEDIR.C	[boot drive]:\DMISL\EXAMPLES\CI	The source file for the direct-interface instrumentation for the sample component.
NAMEDIR.DEP	[boot drive]:\DMISL\EXAMPLES\CI	The dependency file for use in compiling the NAMEDIR.C file. You might need to modify this file to accommodate path differences in your system.
NAMEDIR.MAK	[boot	The make file for use in com-

```
                drive]:\DMISL\EXAMPLES\CI  piling and linking the sample
                                           component instrumentation
                                           (NAMEDIR.C). You can use this
                                           make file with the NMAKE utility
                                           provided by version 2 of the IBM
                                           C/C++ Tools product. You might
                                           need to modify this file to
                                           accommodate path differences in
                                           your system.

NAMEDIR.MIF     [boot                      The sample MIF file that defines
                drive]:\DMISL\EXAMPLES\CI  the NAMEDIR component.
```

Implementation Considerations for OS/2

This section describes characteristics of the service layer that you need to be aware of when developing applications and instrumentation in an OS/2 environment.

16-Bit to 32-Bit Thunking

The service layer is a 32-bit application. If you are developing component instrumentation or a management application that is a 16-bit application, ensure that your application supports 16-bit to 32-bit thunking.

Callback Threading

The confirm callback that the service layer returns to an application is sent on a separate thread from that used by the application's executable file. The application can post a message directly on the callback.

Ensure that your application does not block the callback thread.

Synchronous and Asynchronous Components

The example components NAMES and NAMEDIR are both designed to respond to requests in a synchronous manner. The component processes each request as it is received and does not process subsequent requests until the previous request has been answered.

You can take a synchronous component and modify it to function asynchronously by doing the following:

- Create a thread that is blocked until a command is received from a management application.
- When the command is received, the component instrumentation queues the command to the waiting thread and returns from the DmiCiInvoke() function call immediately. The service layer continues operation and does not call the component again until the first command has completed.
- The thread processes the command just as it would in a synchronous situation, but after the thread issues the callback to the service layer, it returns to the blocked state or terminates.

This enables the service layer to process other requests.

Debugging Overlay Instrumentation

Overlay component instrumentation can be difficult to debug because of its transitory nature. To facilitate this kind of debugging, you can use a debug option when running the service layer.

Before you use this option, you must stop the service layer, if it's already running. If you start the service layer automatically when your system starts, do the following:

- Comment out the `RUN=` line in your CONFIG.SYS file that starts the DMISL.EXE program.
- Restart your system.

To start the service layer with the debug option, enter the following command from an OS/2 command prompt:

```
DMISL DEBUG
```

If the subdirectory containing the service layer executable file is not in your PATH statement, enter the appropriate drive and path as part of the command.

When the service layer is loaded, a debug address is displayed. When you set a breakpoint at this address, execution stops just before the DmiCiInvoke() function call is issued. At this point the overlay component DLL is already loaded, and you can specify breakpoints in the instrumentation wherever you need them.

Service Layer Time-out

If the OS/2 service layer provided with SystemView Agent issues a request to a component and the component takes longer than 15 seconds to respond, the service layer automatically deregisters the component. The service layer then terminates any other outstanding requests to the component. Further requests sent to the component yield an error (`RC=1003; CPERR_CI_TERMINATED`).

The DMI browser continues to display the component icon until the component is deinstalled. If you attempt to expand the icon after the time-out has occurred, the DMI browser returns an error.

Implementing DMI on AIX

This chapter describes the AIX files provided with the SystemView Agent program for use in creating applications and component instrumentation in an AIX environment. The following types of files are included:

- Program files related to the service layer
- Example files for an AIX management application and component, including overlay and direct-interface instrumentation for the component

The chapter also presents any implementation considerations that are specific to AIX.

Program Files

AIX Program Files displays a list of AIX program files provided with the SystemView Agent program. The subdirectory indicating a file's location is the default subdirectory used during installation. If you installed SystemView Agent in a different directory, adjust the file location accordingly.

Table 55. AIX Program Files

FILE NAME	LOCATION	DESCRIPTION
dmisl	/usr/lpp/sva/bin	The service layer executable file. You must start this program before any applications can access the service layer.
dmisl.cat	/usr/lpp/sva/nls/en_US	The message file containing all messages associated with the service layer.
dmix.cat	/usr/lpp/sva/nls/en_US	The message file containing the messages associated with the DMI browser.
dmi.base.smt	/usr/lpp/sva/nls/en_US	The message file containing the messages associated with SMIT.
dmix	/usr/lpp/sva/bin	The executable file for the DMI MIF browser.
dmi.h	/usr/lpp/sva/include	The header file for the service layer.
dmiapi.h	/usr/lpp/sva/include	The header file for the DMI procedure library. Refer to DMI Procedure Library (DMIAPI) for more information about the DMI procedure library.
os_dmi.h	/usr/lpp/sva/include	The header file for the AIX operating system.
error.h	/usr/lpp/sva/include	The header file for the service layer's error message file (dmisl.cat).
libdmisl.a	/usr/lpp/sva/lib	DMI import library for use when compiling applications. Refer to DMI Procedure Library (DMIAPI) for more information about the DMI procedure library.
libdmiapi.a	/usr/lpp/sva/lib	DMI import library for use when compiling applications that are not X-Windows applications. Refer to DMI Procedure Library (DMIAPI) for more information about the DMI procedure library.
libdmiapiX.	/usr/lpp/sva/lib	The DMI import library for use when compiling applications that are X-Windows applications. Refer to DMI Procedure Library (DMIAPI) for more information about the DMI procedure library.

Example Files

AIX Example Files displays a list of AIX example files provided with SystemView Agent. The subdirectory indicating a file's location is the default subdirectory used during installation. If you installed SystemView Agent in a different directory, adjust the file location accordingly.

Table 56. AIX Example Files

FILE NAME	LOCATION	DESCRIPTION
miexamp.c	/usr/lpp/sva/samples	The source code used by the DMI MIF browser to interact with the Management Interface.
names.c	/usr/lpp/sva/samples	The source file for the overlay instrumentation for the sample component.
Makefile.sample	/usr/lpp/sva/samples	The make file for use in compiling and linking the sample component instrumentation (names.c). You might need to modify this file to accommodate path differences in your system.
names.mif	/usr/lpp/sva/samples	The sample MIF file that defines the NAMES component. You might need to modify the path definition in the file to accommodate path differences in your system.
namedir.c	/usr/lpp/sva/samples	The source file for the direct-interface instrumentation for the sample component.
namedir.mif	/usr/lpp/sva/samples	The sample MIF file that defines the NAMEDIR component.

Implementation Considerations for AIX

This section describes characteristics of the service layer that you need to be aware of when developing applications and instrumentation in an AIX environment.

Synchronous and Asynchronous Components

The example components NAMES and NAMEDIR are both designed to respond to requests in a synchronous manner. The component processes each request as it is received and does not process subsequent requests until the previous request has been answered.

You can take a synchronous component and modify it to function asynchronously by doing the following:

- Create a thread that is blocked until a command is received from a management application.

- When the command is received, the component instrumentation queues the command to the waiting thread and returns from the DmiCiInvoke() function call immediately. The service layer continues operation and does not call the component again until the first command has completed.
- The thread processes the command just as it would in a synchronous situation, but after the thread issues the callback to the service layer, it returns to the blocked state or terminates.

This enables the service layer to process other requests.

Service Layer Time-out

If the AIX service layer provided with SystemView Agent issues a request to a component and the component takes longer than 15 seconds to respond, the service layer automatically deregisters the component. The service layer then terminates any other outstanding requests to the component. Further requests sent to the component yield an error (`RC=1003; CPERR_CI_TERMINATED`).

The DMI browser continues to display the component icon until the component is deinstalled. If you attempt to expand the icon after the time-out has occurred, the DMI browser returns an error.

Compiling Overlay Components

If you are developing an overlay component in AIX, use the `-e DmiCiEntryPoint` option when linking the program. This flag changes the entry point of the component from "main" to "DmiCiEntryPoint". DmiCiEntryPoint is a function that provides the service layer with the function addresses of DmiCiInvoke() and DmiCiCancel(). The service layer needs these addresses to communicate with the overlay component and requires DmiCiEntryPoint to be the entry point of the overlay component because of programming considerations in AIX.

The following example indicates how these flags can be used in an AIX makefile:

```
names:  names.o Makefile $(SLLIBS)
       $(CCC) -o $@ -e DmiCiEntryPoint names.o $(LINKLIBDIR) $(LIBS)
```

Security Considerations with Overlay Components

Because overlay component instrumentation code runs as part of the service layer and is executed with root privileges, these components must be installed in the /usr/lpp/sva/overlay directory. You must have root privileges to write to this directory.

Changing the privileges of the /usr/lpp/sva/overlay directory could allow a non-root user to install a component that would then run with root privileges, creating a potential security exposure. For this reason, it is important that you ensure that the /usr/lpp/sva/overlay directory is only writeable by those with root privileges.

Implementing DMI on Windows NT/Windows 95

This chapter describes the Win32 files provided with the SystemView Agent program for use in creating applications and component instrumentation in a Windows NT or Windows 95 environment. The following types of files are included:

- Program files related to the service layer
- Example files for a Windows NT/Windows 95 management application and component, including overlay and direct-interface instrumentation for the component

The chapter also presents any implementation considerations that are specific to Windows NT/Windows 95.

Program Files

Windows NT/Windows 95 Program Files displays a list of Windows NT/Windows 95 program files provided with the SystemView Agent program. **C:\SVA** is the default subdirectory used during installation. If you installed SystemView Agent in a different drive or directory, adjust the file location accordingly.

Table 57. Windows NT/Windows 95 Program Files

FILE NAME	LOCATION	DESCRIPTION
DMISLSRV.EX	"C:\SVA"\DMI\BIN	The service layer executable file for Window NT. You must start this program before any applications can access the service layer.
DMISLAPP.EX	"C:\SVA"\DMI\BIN	The service layer executable file for Window 95. You must start this program before any applications can access the service layer.
DMIAPI32.DL	"C:\SVA"\DMI\BIN	The dynamic link library (DLL) for the DMI procedure library. The SystemView Agent installation process adds this subdirectory to the PATH statement.
DMIWIN32.EX	"C:\SVA"\DMI\BIN	The executable file for the DMI MIF browser.
DMI.H	"C:\SVA"\DMI\INCLUDE	The header file for the service layer.
DMI_API.H	"C:\SVA"\DMI\INCLUDE	The header file for the DMI procedure library. Refer to DMI Procedure Library (DMIAPI) for more information about the DMI procedure library.
OS_DMI.H	"C:\SVA"\DMI\INCLUDE	The header file for the Windows NT and Windows 95 operating systems.
ERROR.H	"C:\SVA"\DMI\INCLUDE	The header file for the service layer's error messages.
DMIAPI32.LI	"C:\SVA"\DMI\LIB	The DMI import library for use with version 2.0 of the Microsoft Visual C++ product. Refer to DMI Procedure Library (DMIAPI) for more information about the DMI procedure library.

Example Files

Windows NT/Windows 95 Example Files displays a list of Windows NT/Windows 95 example files provided with SystemView Agent. **C:\SVA** is the default subdirectory used during installation. If you installed SystemView Agent in a different drive or directory, adjust the file location accordingly.

Table 58. Windows NT/Windows 95 Example Files

FILE NAME	LOCATION	DESCRIPTION
DMI32.C	"C:\SVA"\DMI\EXAMPLES\MI	The source code used by the DMI MIF browser to interact with the Management Interface.
NAMES.C	"C:\SVA"\DMI\EXAMPLES\CI	The source file for the overlay instrumentation for the sample component.
NAMES.DEF	"C:\SVA"\DMI\EXAMPLES\CI	The module definition file for use in building the DLL file for the overlay instrumentation. The NAMES.DEF file defines the entry points and other characteristics of the DLL.
NAMES.MAK	"C:\SVA"\DMI\EXAMPLES\CI	The make file for use in compiling and linking the sample component instrumentation (NAMES.C). You can use this make file with version 2.0 of the Microsoft Visual C++ product. You might need to modify this file to accommodate path differences in your system.
NAMES.MIF	"C:\SVA"\DMI\EXAMPLES\CI	The sample MIF file that defines the NAMES component. You might need to modify the path definition in the file to accommodate path differences in your system.
NAMEDIR.C	"C:\SVA"\DMI\EXAMPLES\CI	The source file for the direct-interface instrumentation for the sample component.
NAMEDIR.MAK	"C:\SVA"\DMI\EXAMPLES\CI	The make file for use in compiling and linking the sample component instrumentation (NAMEDIR.C). You can use this make file with version 2.0 of the Microsoft Visual C++ product. You might need to modify this file to accommodate path differences in your system.
NAMEDIR.MIF	"C:\SVA"\DMI\EXAMPLES\CI	The sample MIF file that defines the NAMEDIR component.

Implementation Considerations for Windows NT/Windows 95

This section describes characteristics of the service layer that you need to be aware of when developing applications and instrumentation in a Windows NT/Windows 95 environment.

16-Bit to 32-Bit Thunking

The service layer is a 32-bit application. If you are developing component instrumentation or a management application that is a 16-bit application, ensure that your application supports 16-bit to 32-bit thunking.

Callback Threading

The confirm callback that the service layer returns to an application is sent on a separate thread from that used by the application's executable file. The application can post a message directly on the callback.

Ensure that your application does not block the callback thread.

Synchronous and Asynchronous Components

The example components NAMES and NAMEDIR are both designed to respond to requests in a synchronous manner. The component processes each request as it is received and does not process subsequent requests until the previous request has been answered.

You can take a synchronous component and modify it to function asynchronously by doing the following:

- Create a thread that is blocked until a command is received from a management application.
- When the command is received, the component instrumentation queues the command to the waiting thread and returns from the DmiCiInvoke() function call immediately. The service layer continues operation and does not call the component again until the first command has completed.
- The thread processes the command just as it would in a synchronous situation, but after the thread issues the callback to the service layer, it returns to the blocked state or terminates.

This enables the service layer to process other requests.

Debugging Overlay Instrumentation on Windows NT

Overlay component instrumentation can be difficult to debug because of its transitory nature. To facilitate this kind of debugging, you can use a debug option when running the service layer on Windows NT. Refer to the *SystemView Agent for Win32 User's Guide* for information on how to stop the service layer.

Because the service layer normally runs as a background process, you must stop the service layer (if it's already running) before you use this option.

To start the service layer with the debug option, enter the following command from a DOS command prompt:

```
START DMISLSRV DEBUG
```

If the subdirectory containing the service layer executable file is not in your PATH statement, enter the appropriate drive and path as part of the command.

When the service layer is loaded, a debug address is displayed. When you set a breakpoint at this address, execution stops just before the DmiCiInvoke() function call is issued. At this point the overlay component DLL is already loaded, and you can specify breakpoints in the instrumentation wherever you need them.

You can display the version of the service layer on Windows NT by issuing the following from a DOS command prompt:

```
DMISLSRV VERSION
```

This displays version information without starting the service layer. You can issue this command while the service layer is running.

Service Layer Time-out

If the Windows NT/Windows 95 service layer provided with SystemView Agent issues a request to a component and the component takes longer than 15 seconds to respond, the service layer automatically deregisters the component. The service layer then terminates any other outstanding requests to the component. Further requests sent to the component yield an error (`RC=1003; CPERR_CI_TERMINATED`).

The DMI browser continues to display the component icon until the component is deinstalled. If you attempt to expand the icon after the time-out has occurred, the DMI browser returns an error.

Notices

Second Edition (September 1996)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM reseller or IBM marketing representative.

Copyright Notices

Disclaimers

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program or service is not intended to state or imply that only that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

Asia-Pacific users can inquire, in writing, to the IBM Director of Intellectual Property and Licensing, IBM World Trade Asia Corporation, 2-31 Roppongi 3-chome, Minato-ku, Tokyo 106, Japan.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Department LZKS, 11400 Burnet Road, Austin, TX 78758 U.S.A. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

Common User Access	DB2/2
DATABASE 2	FFST/2
First Failure Support Technology/2	IBM
Operating System/2	OS/2
SystemView	

The following terms are trademarks of other companies:

Term	**Trademark of**
DMI	Desktop Management Task Force
DMTF	Desktop Management Task Force
Windows NT	Microsoft Corporation
Win32	Microsoft Corporation
X-Windows	Massachusetts Institute of Technology

Microsoft, Windows and the Windows 95 Logo are trademarks of Microsoft Corporation.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Glossary

This glossary includes terms and definitions from:

- The *American National Standard Dictionary for Information Systems* , ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.
- The ANSI/EIA Standard-440-A, *Fiber Optic Terminology* . Copies may be purchased from the Electronic Industries Association, 2001 Pennsylvania Avenue, N.W., Washington, DC 20006. Definitions are identified by the symbol (E) after the definition.
- The *Information Technology Vocabulary* , developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.
- The *IBM Dictionary of Computing* , New York: McGraw-Hill, 1994.
- Internet Request for Comments: 1208, *Glossary of Networking Terms* .
- Internet Request for Comments: 1392, *Internet Users' Glossary* .
- The *Object-Oriented Interface Design: IBM Common User Access Guidelines* , Carmel, Indiana: Que, 1992.

The following cross-references are used in this glossary:

Contrast with:	This refers to a term that has an opposed or substantively different meaning.
Synonym for:	This indicates that the term has the same meaning as a preferred term, which is defined in its proper place in the glossary.
Synonymous with:	This is a backward reference from a defined term to all other terms that have the same meaning.
See:	This refers the reader to multiple-word terms that have the same last word.
See also:	This refers the reader to terms that have a related, but not synonymous, meaning.
Deprecated term for:	This indicates that the term should not be used. It refers to a preferred term, which is defined in its proper place in the glossary.

A

---

agent

agent

1. As defined in the SNMP architecture, an agent or an SNMP server is responsible for performing the network management functions requested by the network management stations.
2. A role that management programs can assume when dealing with managed objects. A management program assuming an agent role receives requests from and sends notifications to managers (management application programs) or other agents.
3. A system that assumes such a role. *System* here can mean an operating system or some other programming hardware, or microcode (including licensed internal code) support that represents a set of resources.
4. A management program that represents one or more managed objects. An agent receives and services requests for operations and attributes, which are defined for the managed objects it represents. Also, an agent is responsible for emitting notifications (special messages) when it detects conditions in the managed object that require attention.
5. In the TCP/IP environment, a process running on a network node that responds to requests and sends information.

---

attribute

attribute

In the DMI, the smallest descriptive element of the Management Information Format. An attribute describes a single characteristic of a component.

C

---

component

component

In the DMI, a physical or logical element of a system, such as a piece of hardware or software.

---

component instrumentation

component instrumentation

In the DMI, the executable code that provides the values for attributes in a component MIF file. Instrumentation can take one of two forms:

- Runtime programs that are run by the service layer at the time a management application makes a request
- Direct-interface programs that run continuously and link to the service layer when a management application makes a request

---

CONFIG.SYS file

CONFIG.SYS file

A file that contains configuration options for an OS/2 program installed on a workstation.

D

Desktop Management Interface (DMI)

Desktop Management Interface (DMI)

A protocol-independent set of interfaces defined by the Desktop Management Task Force (DMTF) to provide management applications with standardized access to information about hardware and software in a system.

Desktop Management Task Force

Desktop Management Task Force

The Desktop Management Task Force (DMTF) is a vendor alliance which was convened to streamline the management of diverse operating systems commonly found in an enterprise. The DMTF includes industry-wide workgroups, which are actively identifying the pieces of information which are necessary to manage specific categories of devices.

DMI

DMI

Desktop Management Interface.

DMTF

DMTF

Desktop Management Task Force.

---

discovery

discovery

The automatic detection of network topology change, for example, new and deleted nodes or interfaces.

---

drive

drive

The device used to read and write data on disks or diskettes.

E

---

event

event

In the DMI, an unsolicited notification sent from a component to the service layer. For example, an event can indicate that an error has occurred or that a new version of a software product has been installed. Events that are forwarded through the service layer to management applications are referred to as **indications**.

G

---

group

group

In the Management Information Format of the DMI, a group is a set of related attributes used in describing a component.

H

---

host name

host name

A unique name, set at the management protocol level, that identifies a node.

I

IBM Operating System/2

IBM Operating System/2

The base operating system for OS/2 programs.

ID

ID

Identification; identifier.

---

instrumentation

instrumentation

In the DMI, the program code that carries out management actions for a component.

K

---

key attribute

key attribute

In the DMI, the attribute whose identifier (ID) is used as an index into a table. The group to which the key attribute belongs represents a row (or instance) of the table.

L

---

load

load

To move data or programs into memory.

M

---

management application program

management application program

A program that manages some aspects of a distributed system by using management protocols to converse with remote systems.

---

management information

management information

Information within an open system that can be transferred by network management protocols.

---

management information base (MIB)

management information base (MIB)

1. A standard used to define SNMP objects, such as packet counts and routing tables, that are in a TCP/IP environment.
2. All of the accessible information on a managed network.
3. A collection of managed-object definitions.
4. A set of variable bindings that reflect the current state of an SNMP agent. Extensions to the MIB can be added by a business enterprise.

Management Information Format

Management Information Format

The format specified by the DMI for defining the aspects of a component that can be managed. Text files that conform to the MIF format are referred to as MIF files.

MIB

MIB

Management information base.

MIF

MIF

Management Information Format

MIF file

MIF file

In the DMI, a text file that conforms to the format of the MIF and describes all aspects of a component that can be managed.

O

---

object ID (OID)

object ID (OID)

The unique name identification of a management information base object.

OID

OID

Object ID.

P

---

process

process

1. A unique, finite course of events defined by its purpose or by its effect, achieved under defined conditions.
2. Any operation or combination of operations on data.
3. A function being performed or waiting to be performed.

4. A program in operation; for example, a daemon is a system process that is always running on the system.
5. An address space, one or more threads of control that run within that address space, and the required system resources.
6. A collection of system resources that include one or more threads of execution that perform a task.

---

protocol

protocol

1. A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication. (I) Protocols can determine low-level details of machine-to-machine interfaces, such as the order in which bits from a byte are sent; they can also determine high-level exchanges between application programs, such as file transfer.
2. A set of rules governing the operation of functional units of a communication system that must be followed if communication is to take place.
3. In Open Systems Interconnection architecture, a set of semantic and syntactic rules that determine the behavior of entities in the same layer in performing communication functions. (T)

Request For Comments (RFC)

Request For Comments (RFC)

A series of documents that covers a broad range of topics affecting internetwork communication. Some RFCs are established as Internet standards.

RFC

RFC

Request for Comments

S

Simple Network Management Protocol (SNMP)

Simple Network Management Protocol (SNMP)

1. A protocol that allows network management by elements, such as gateways, routers, and hosts. This protocol provides a means of communication between network elements regarding network resources.

2. A protocol running above TCP/IP that is used to exchange management information.
3. A protocol running above the User Datagram Protocol (UDP) used to exchange network management information.

SNMP

SNMP

Simple Network Management Protocol.

---

system

system

1. A computer and its associated devices and programs.
2. An end point of a communications link or a junction common to two or more links in a network. Systems can be processors, communication controllers, cluster controllers, terminals, workstations, clients, requesters, or servers.

T

TCP/IP

TCP/IP

Transmission Control Protocol/Internet Protocol.

---

topology

topology

The schematic arrangement of the links and nodes of a network.

Transmission Control Protocol/Internet Protocol (TCP/IP)

Transmission Control Protocol/Internet Protocol (TCP/IP)

A set of communication protocols that supports peer-to-peer connectivity functions for both local and wide-area networks.

---

trap

trap

1. An unprogrammed conditional jump to a specified address that is automatically activated by hardware. A recording is made of the location from which the jump occurred. (I)

2. An unsolicited event generated by an agent and forwarded to a manager. Traps inform the manager of changes that occur in the network.

U

UDP

UDP

User Datagram Protocol.

User Datagram Protocol (UDP)

User Datagram Protocol (UDP)

In IP, a packet-level protocol built directly on the Internet protocol layer. UDP is used under SNMP for application-to-application programs between IP host systems.

Bibliography

This bibliography contains a list of publications pertaining to the product and related subjects.

SystemView Agent Publications

The following paragraphs briefly describe the library for Version 1 of the SystemView Agent program:

SystemView Agent User's Guide (SVAG-USR2)

This book provides instructions for installing the SystemView Agent package and using the MIF browser and describes communication with SNMP management applications.

SystemView Agent DMI Programmer's Guide (SVAG-DMIP)

This books describes the operation of the DMI, including the command blocks and the conventions of the MIF. The book also discusses how to enable components for DMI access and how DMI information is made available to SNMP management applications.

SystemView Agent DPI Programmer's Guide (SVAG-DPIP)

This book describes the distributed protocol interface (DPI), including programming concepts, basic API functions, and examples.

Desktop Management Task Force Publications

The following publications are available from the Desktop Management Task Force:

Desktop Management Interface Specification, Version 1.1

Internet Request for Comments Documents

The Internet protocol suite is still evolving through Requests for Comments (RFCs). New protocols are being designed and implemented by researchers and are brought to the attention of the Internet community in the form of RFCs.

As networks have grown in size and complexity, SNMP has emerged as the Internet network management standard. The following RFCs provide information relevant to SNMP and related to SystemView Agent:

RFC 1155: Structure and Identification of Management Information for TCP/IP-based Internets

This RFC provides a set of rules used to define and identify MIB objects.

RFC 1157: A Simple Network Management Protocol (SNMP)

This RFC defines the protocol used to manage objects defined in a MIB.

RFC 1212: Concise MIB Definitions

RFC 1213: Management Information Base for Network Management of TCP/IP-based Internets: MIB-II

This RFC defines a base set of managed objects to be provided in a MIB.

RFC 1592, SNMP Distributed Protocol Interface (DPI), Version 2.0

This RFC describes the protocol to allow an SNMP agent to communicate with a subagent. This allows users to dynamically add, delete, or replace MIB objects without recompiling the SNMP agent.

RFC 1901, Introduction to Community-based SNMPv2

RFC 1902, Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1903, Textual Conventions for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1904, Conformance Statements for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1905, Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1906, Transport Mappings for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1907, Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1908, Coexistence between Version 1 and Version 2 of the Internet-standard Network Management Framework

RFC 1909, An Administrative Infrastructure for SNMPv2
